The C(99) Programming Language

Reminders & Advanced Features
The C99 Standard

- Standards define how a programming language behaves
  - Syntax
  - The standard library

- Writing according to standards is important:
  - Ensures things behave as expected
  - Allows porting code to different compilers and systems

- C99 is the C standard from 1999
  - Has several additions over previous standards
  - C11 is a newer standard, but not relevant to this course
Variables can be defined **anywhere**!

- Define a variable **only** when it is needed and can be assigned a **meaningful value**

```c
int main() {
    printf("Please enter your favorite number: ");

    unsigned num;
    scanf("%u", &num);

    for (int i = 0; i < num; ++i) {
        printf("%u is a great choice!", num);
    }
    return 0;
}
```
Array sizes can be **determined at run-time**
- Dynamic allocation is still needed to **allocate large arrays**, to exchange **varying data across functions and modules**, or to **control when the data is freed**

```c
int main() {
    printf("Please a number: ");

    unsigned num;
    scanf("%u", &num);

    double num_array[num];

    return 0;
}
```

will crash if `num` is too big (allocated on the stack, that is relatively small)!
C99 defines a `bool` type
- Can be either `true` or `false`
- Defined in `<stdbool.h>`

```c
#include <stdbool.h>
#include <math.h>

bool areClose(double a, double b, double epsilon) {
    return fabs(a - b) < epsilon;
}
```

What is the advantage of using `true` & `false` over `1` & `0`?
- Makes the code clearer
The `enum` keyword defines a new type and a set of named values it can take:
- Each name is associated with an integral value
- The names are more important than the actual values
- By default start with zero and increment by one

```c
// definitions
enum Gender { MALE, FEMALE };  
enum Month { JAN, FEB, MAR, /* ... */ , DEC, MONTHS_NUM };  
enum Season { SUMMER=1, FALL, WINTER=8, SPRING };  

// usage
enum Gender gender = MALE;  
enum Season seasons[MONTH_NUM];  

season[JAN] = WINTER;  

enum Month next_month(enum Month current) {
  if (current == DEC)
    return JAN;
  return (enum Month)(current + 1);
}
```
enum DisplayMode { SILENT, NORMAL, VERBOSE };  
enum ErrorFlag { FAILURE, SUCCESS };  

// find extremum point of ax^2+bx+c  
enum ErrorFlag find_optimum(double a, double b, double c, double* x, enum DisplayMode mode)  
{  
    if (fabs(a)<1e-8) {  // error: linear function  
        switch (mode) {  
            case VERBOSE:  
                fprintf(stderr,"Error: no optimum point in %lf*x^2 + %lf*x + %lf\n", a, b, c);  
                break;  
            case NORMAL:  
                fprintf(stderr,"Error: no optimum point\n");  
                break;  
        }  
        return FAILURE;  
    }  
    *x = -b/(2*a);  
    return SUCCESS;  
}  

double x0;  
if (find_optimum(1.0, -3.0, 4.0, &x0, SILENT) == SUCCESS)  
    printf("polynomial extremum at: %lf\n", x0);  

Introduction to Systems Programming
The **const** keyword

```c
const type identifier = expression;  // cannot change after initialization
```

```c
const double PI = 3.141592654;  
const double EPSILON = 1e-6;  
double rad;
```

```c
if (PI*rad*rad < EPSILON) {
    printf("circle is too small!\n");
}
```

- **const** versus **enum**
  - Can represent non-integer values
  - A regular variable that has memory alloc. (and hence “&PI” is valid)

- **const** versus **#define**
  - #define is processed at cpp/compilation time
  - A const variable get its value at runtime (i.e. function call)
  - A const variable is scoped (like all C variables)

Time to think **when** things happen: cpp, compile, link, run time
A normal function call has a small overhead (managing the stack, copying parameter values, etc.)

A small overhead can become large if function is critical

```c
int max(int a, int b) {
    return a > b ? a : b;
}

int array_max(int a[], int n) {
    int result = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] > result)
            result = a[i];
    }
    return result;
}
```

this code is clearer

```c
int array_max(int a[], int n) {
    int result = a[0];
    for (int i = 1; i < n; ++i) {
        if (a[i] > result)
            result = a[i];
    }
    return result;
}
```

but this will be significantly faster for a large array
Inline Functions (C99)

- Indicating a function as `inline` asks the compiler to replace each call to this function with efficient equivalent code that does not perform a function call:

```c
inline int max(int a, int b) {
    return a > b ? a : b;
}
```

```c
int a,b,c,d;
...
int x = max(a,b);
int y = max(c,d);
int z = max(3,4);
```

if `max()` is inlined, the compiler is asked to compile code equivalent to this:

```c
int x = a > b ? a : b;
int y = c > d ? c : d;
int z = 4;
```

Note this one
Using inline functions produces **larger** compiled files
- Code will be replicated in every call
- Use **inline** with **short** and **critical** functions only

The inline keyword is just a **recommendation**
- The compiler may choose **not** to inline a function (e.g., it is too long)
- The compiler may choose to inline a function even if it is not indicated as inline, when optimizations are enabled

It is hard to **predict** whether inlining a function will improve efficiency or not
- Use a **profiler** instead of guesswork
- Avoid optimizations if they are **not actually needed**
The **include** directive causes the *cpp* (C preprocessor) to copy and paste the content of the included file.

**function.h**

```c
int function1(int n);
int function2(int n);
```

**main.c**

```c
#include "function.h"

int main() {
    printf("%d\n", function1(5));
    return 0;
}
```

**gcc -E main.c**

```c
int function1(int n);
int function2(int n);

int main() {
    printf("%d\n", function1(5));
    return 0;
}
```

only preprocess the code
A typical project has hundreds (or more) of functions, types, and constants.

We typically want to separate our code into multiple files:
- Makes it easier to find code which is relevant to a specific task
- Makes it easier to cooperate on a project

```c
int max(int a, int b) {
    return a > b ? a : b;
}

int array_max(int a[], int n) {
    int result = a[0];
    for (int i = 1; i < n; ++i) {
        result = max(a[i], result);
    }
    return result;
}

void print_array(int a[], int n) {
    ...
}

void run_test_1() {
    int a[] = { 3, 4, 1, 5, 6 };  
    int maxval = array_max(a, sizeof(a) / sizeof(int));
    print_array(a);
    ...
}

void run_test_2() {
    ...
}

int main() {
    run_test_1();
    run_test_2();
    return 0;
}
```
Several files can be easily compiled into one executable file:

```bash
> gcc main.c utility.c testing.c
```
don’t forget to add flags

In order to call the function `f` defined in file `a.c` from a different file `b.c`, `f` must be declared in `b.c`

```
int array_max(int a[], int n) {
    int result = a[0];
    for (int i = 1; i < n; ++i) {
        result = max(a[i], result);
    }
    return result;
}
```

we’ll soon replace this with `#include`

In order to use a type, it must be defined.

```
int array_max(int a[], int n);
```

```
void run_test_1() {
    int a[] = { 3, 4, 1, 5, 6 };
    int maxval = array_max(a, sizeof(a) / sizeof(int));
    print_array(a);
    ...
}
```
The **Compilation Process**

- **User Header**: 
  - \*.h

- **Source Code**: 
  - \*.c

- **Standard Library Headers**: 
  - \*.h

- **Processed Code**: 
  - \*.c

- **C Preprocessor (aka cpp)**
  - \*.h

c
  - \*.c

- **C Compiler**
  - \*.c

- **Object File**: 
  - \*.o/*.*.obj

- **Linker**
  - \*.a/*.*.lib

- **(Static) Libraries**
  - \*.a/*.*.lib

- **Executable**
  - \*/*.*.exe

**Unix/Windows convention**
An object file contains compiled (machine) code of a single source file
- And contains entry locations/symbol tables, debugging information, etc.
- An object file cannot be executed since it may contain (and hence depend on) calls to functions/variables defined in other files

The “-c” flag makes gcc stop after creating object files

```bash
> gcc -c utility.c testing.c
```

* Binary code is for illustration purposes and is not real machine code
The **linking** phase takes several object files and libraries as input and merges them into a single executable file.

- Associates each function call/variable use with the code/data location.
- Each function/variable must be defined in exactly one object file or in at least one library.
  - Functions/variable found in an object file take precedence over a library.
  - Order of libraries matter: first library found to have function/variable wins.
- A main() function must be defined.
  - This function is called when the program starts.

```bash
> gcc utility.o testing.o main.o
```

This machine code now means “call the function located at address XXXX, with parameters a and 5”.
Compilation and linking can be combined. In this case, every .c file is first compiled to a .o, and then the linker is immediately invoked:

```bash
> gcc main.c utility.c testing.c
```

- Creating an executable called ‘a.out’ by default.

Different combinations are possible (.o taken verbatim)...

```bash
> gcc -o test main.c utility.o testing.o
```

- Creating an executable called ‘test’.

gcc (like other compilers) has flags to perform only specific actions. I.e.:

- `-E`: Stop after preprocessing
- `-c`: Stop after compilation (only create object files)
- `-o name`: specifies the output (object or executable) name.
Recompiling an entire project can take many hours.

By using selective compilation we can avoid most of the time needed for recompiling:
- Only the object files that have changed need to be recompiled
- Linking still needs to be done, but this is faster
  - Some modern linkers offer global optimizations that requires knowledge on the entire project.

As a project gets bigger, tools are needed to manage its dependencies and modular compilation:
- The tool called make is an example (more in tutorials)
The C Preprocessor

- User Header
- Standard Library Headers
- Processed Code
- Object File
- Static Library
- Executable

Unix/Windows convention
The C Preprocessor

- **File Inclusion**
  
  ```
  #include <file-name>
  #include "file-name"
  ```

- **Macro definitions**
  
  ```
  #define identifier value
  #define identifier(variables) value
  #undef identifier
  ```

- **Conditional compilation**
  
  ```
  #if condition
  ifndef identifier
  ifndef identifier
  #else
  #endif
  ```
C Preprocessor: Macros

- The preprocessor replaces each appearance of `identifier` in the code by the `token-sequence` (merely text substitution)
  - If `parameters` are involved, they are replaced with the `arguments`
  - No spaces are allowed between the identifier and the parentheses

- If needed, macros can be undefined:

  `#undef identifier`

- A macro is active from the `line after its declaration` and until the end of the file or (and if) it is `undefined` or `redefined`
  - `#define` an already defined macro without `#undef` is considered a cpp error.
C Preprocessor: Macros

- The preprocessor **does not know C**. It replaces the text **exactly** as it is directed. In essence a **text editor**.
- If the result is not legal C code, the compilation will **fail**.

```c
#define return :)  

int main() { 
    return 0;
}  
```
Macros can be used to define constants

- Replace **magic values** with a **meaningful name**

```c
#define MAX_STR_LEN 20
...
// Usage:
char arr[MAX_STR_LEN + 1];
...
for (char* str = arr; *str != 0; str++) {
    if (*str >= 'A' && *str <= 'Z')
        *str = *str - 'A' + 'a';
}
```

```c
...
// Usage:
char arr[20 + 1];
...
for (char* str = arr; *str != 0; str++) {
    if (*str >= 'A' && *str <= 'Z')
        *str = *str - 'A' + 'a';
}
```
Macros with Parameters: First Attempt

```c
#define SQR(x) x*x
...
// Usage:
double res1 = SQR(5);
double res2 = SQR(5+3);
```

Preprocessing

Uh-oh....

```c
double res1 = 5*5;
double res2 = 5+3*5+3;
```
Several **problems** may arise while calling a macro

```c
#define SQR(x) x * x
```

- **Operator precedence errors**
  - **External**
    ```c
    (int)SQR(a) // is expanded to -> (int)a * a // which casts the left operand only
    ```
  - **Internal**
    ```c
    SQR(a + b) // is expanded to -> a + b * a + b // which computes the wrong thing
    ```
  - **Solution: Use parenthesis**
    ```c
    #define SQR(x) ((x) * (x))
    ```
    - This is always safer (but not always safe...)

Introduction to Systems Programming
Several additional problems may arise, calling a macro

```c
#define SQR(x) ((x) * (x))
```

- **Side effect errors**

```c
SQR(++i) // is expanded to ->
((++i) * (++i)) // which may increment i twice
// (and the result is undefined)
```

- **Unnecessary function calls**

```c
SQR(very_difficult_func(a,z,t))
/* will compute the difficult function twice */
```

- **These problems have no solution!** (with macros)

Be wise when **defining** a macro & be cautious when **calling** a macro
```c
#define MAX_STR_LEN 20
#define IS_UPPER(c) ((c) >= 'A' && (c) <= 'Z')
#define TO_LOWER(c) (IS_UPPER(c)? (c) - 'A' + 'a' : (c))
#define SQR(x) ((x) * (x))

A better way to implement IS_UPPER, TO_LOWER and SQR is using
**inline functions**. This has the same efficiency, and **solves the issues**
in the previous slide:

```c
inline double sqr(double x) {
    return x*x;
}
```

**Question:** What assumption do we have here?

- `X = SQR(++i);`  
  *error* (macro may increment `i` twice)
- `X = sqr(++i);`  
  *no problem* (inline function)
Macros vs. Inline functions

- Prefer using functions whenever possible
- When do we still need **macros**?
  - Operation needs to work on a variety of different types:
    ```
    #define MAX(a, b) ((a) > (b) ? (a) : (b))
    #define SWAP(type, a, b) { type t = a; a = b; b = t; }
    ```
  - Some unique features of the preprocessor are needed:
    ```
    #ifdef NDEBUG
    #define assert(x) ((void)0)
    #else /* debugging enabled */
    void _assert (const char*, const char*, int);
    #define assert(e) ((e) ? (void)0 : _assert(#e, __FILE__, __LINE__))
    #endif /* NDEBUG */
    ```
    Question: What is **NDEBUG**?
  - Managing platform specific functionality (portability).

C++ has a better solution for these!
A macro with argument X can use the syntax `#X` to produce a string literal containing the parameter:

```c
#define PRINT_INT(a) printf("%s = %d", #a, a)
```

```c
int x = 5, y = 3;
PRINT_INT(x);
PRINT_INT(y);
PRINT_INT(2*x);
```

And there is also a `##` operator:

```c
int x = 5, y = 3;
printf("%s = %d", "x", x);
printf("%s = %d", "y", y);
printf("%s = %d", "2*x", 2*x);
```

The alternative with inline functions is less convenient:

```c
inline void print_int(int a, char* name) {
    printf("%s = %d", name, a);
}
```

```c
print_int(x, "x");
print_int(y, "y");
print_int(2*x, "2*x");
```
The preprocessor can handle simple conditional statements
- If the condition holds, the text is kept in the preprocessed file, otherwise it is removed
- Remember portability across different platforms?

```c
#define LOW 1
#define MEDIUM 2
#define HIGH 3
#define LOGGING_LEVEL HIGH

void load_configuration() {
    ...
    #if LOGGING_LEVEL > MEDIUM
        printf("Configuration loaded\n");
    #endif
}
```

macros can use other previously defined macros

```c
void load_configuration() {
    ...
    printf("Configuration loaded\n");
}
```
The conditional statement can also have an `#else` clause with alternative code to include if the condition fails.

```c
#define LOW 1
#define MEDIUM 2
#define HIGH 3

#define LOGGING_LEVEL HIGH

#if LOGGING_LEVEL >= MEDIUM
#define LOG_MEDIUM(message) printf(message)
#else
#define LOG_MEDIUM(message) ((void)0)
#endif

void load_configuration() {
    ...
    LOG_MEDIUM("Configuration loaded\n");
}
```

A statement that does nothing (but still must end with a `;`) can do too:

```c
#define LOG_MEDIUM(message) ((void)0)
```
### Conditionals

- Conditional statements can check if a macro has been defined using `defined(identifier)`
  - This is the most common use, and abbreviations are defined for this:

  ```c
  #ifndef MY_MACRO
  #if !defined(MY_MACRO)
  #elif defined(MY_MACRO)
  #endif
  #ifdef MY_MACRO
  #endif
  ```

  What is this? You met it before!

- `Identifier` can also be a whole Boolean expression

  ```c
  #ifdef DEBUG_ON
  #define DEBUG_PRINT(...) fprintf (stderr, __VA_ARGS__)
  #else
  #define DEBUG_PRINT(...) ((void)0)
  #endif
  ```

So why both `#ifdef` and `#if defined`?!
Conditionals can help us add **extra checks while debugging**, without affecting the performance of the final product.

- Remember when is the preprocessor invoked.

```c
int safe_get(struct array arr, int i) {
  #ifdef DEBUG
    if (arr.a == NULL) {
      fprintf(stderr, "Null array\n");
      exit(1);
    }
    if (i < 0 || i >= arr.n) {
      fprintf(stderr, "Out of bounds\n");
      exit(1);
    }
  #endif
  return arr.a[i];
}
```

```c
struct array {
  int* a;
  int n;
};
```

Without **DEBUG** all checks from compiled code are removes

```c
int sum = 0;
for (int i = 0; i < n; i++) {
  sum += safe_get(arr, i);
}
```

**Question:** What is **DEBUG**? **NDEBUG**?
Some special macros are predefined, for example:
- `__LINE__` - expands to the line number in the file being processed
- `__FILE__` - expands to the name of the file being processed

Some macros are defined by the **compiler**
- Allows code to compile under different environments/platforms

```c
#ifdef _WIN32
#include <windows.h>
#define PRINT_ERROR(message) MessageBox(NULL, message, "Error", MB_ICONERROR)
#else
#define PRINT_ERROR(message) fprintf(stderr, message);
#endif

PRINT_ERROR("File not found");
```
The **assert** macro

- A good example for using the preprocessor
  - Used to help find errors as early as possible
  - Defined in `<assert.h>`

```c
#ifdef NDEBUG
#define assert(x) ((void)0)
#else /* debugging enabled */
void _assert (const char* condition, const char* filename, int line);
#define assert(e) ((e) ? ((void)0 : _assert(#e, __FILE__, __LINE__)))
#endif /* NDEBUG */
```

prints an error message and calls abort()

```c
#include <assert.h>
...
int safe_get(struct array arr, int i) {
    assert(arr.a != NULL);
    assert(i >= 0 && i < arr.n);
    return arr.a[i];
}
```

```c
#include <assert.h>
...
int sum(int* array, int size) {
    assert(array != NULL && size >= 0);
    ...
}
```
Many declarations must not appear more than once
- Must therefore protect against including a file more than once

```c
#include "utility.h"
...
#endif /* UTILITY_H */
```

```c
#include "utility.h"
#include "testing.h"
// will not include utility.h again
...
```

```c
#include "utility.h"
#include "testing.h" // will not include utility.h again
...
```

```c
// conditional inclusion of this file
#ifndef TESTING_H
#define TESTING_H
#include "utility.h"

// the contents of testing.h comes here
...
#endif /* TESTING_H */
```

```c
#ifndef TESTING_H
#define TESTING_H
#include "utility.h"

// the contents of testing.h comes here
...
#endif /* TESTING_H */
```

Question: What will happen if utility.h will also “#include testing.h”?
The `typedef` keyword allows declaring a new name for a type.

Syntax: a typedef line looks the same as a variable declaration line, except for the `typedef` word at the beginning.

- Each "variable" declared in this way acts as a nickname to the type it was defined as.

A typedef will commonly appear in a header (.h) file so it can be shared by many code files.

```c
typedef int Length;  // defining

#include "types.h"
...
Length len;          // equivalent to: int len;
Length len_arr[SIZE]; // equivalent to: int len_vec[SIZE]
```
Uses of `typedef`

- Defining a type that may need to be changed in the future

```c
typedef double Real;  // represents real numbers in a physics simulation
                      // if we run into memory problems, change to float
```

- Improve code readability

```c
Length l1, l2;  // we understand l1 and l2 represent lengths
```

- Hide implementation details

```c
#include <stdio.h>

FILE* in_file = fopen("params.dat");  // implementation of the FILE type
                                      // is irrelevant - it is only used
                                      // through dedicated functions
```
Uses of `typedef`

- **Abbreviate long type names**

  ```c
  typedef enum GenderType Gender;  // can now use Gender instead of enum GenderType
  ```

- **Example: define a 3D point type for a physics simulation**

  ```c
  typedef Real Point3d[3];  // Point3D is an array of 3 doubles
  ```

  ```c
  Point3D points[N];  // define an array of N points
  // equivalent to: double points[N][3]
  ```

- **Very Important**: Avoid simple names in global name spaces.
Programming with basic C types can be counter-intuitive when more complex objects and concepts are involved.

Structures are the basis for defining more complex types in C to represent complicated objects.

The `struct` keyword defines a new type which is a composition of several fields:
- Each field has a type and a name.
- Structures can be nested.
- Fields are accessed by the '.' or '->' operators.

**Syntax:**
```
struct [name] { fields-list };
```
Structured can be nested

Assignment between structures is allowed
  - What is the meaning!?

Operators ‘.’ and ‘->’ (same, yet highest, priority).

```c
Struct student st1, st2, *st_ptr;
double grades[5];

st1.name = "my student";
st1.grades = grades;
st1.numGrades = 5;
st2 = st1;
  .
  .
  .
st_ptr = &st1;

if (strcmp(st_ptr -> name, st2.name)
    st_ptr -> name = st2.name;
```

```
struct student {
    char* name;
    double* grades;
    int numGrades;
};
typedef struct student Student;
```
Example: Complex Numbers

```c
struct complex {
    double re, im;  //real and imaginary parts
};
typedef struct complex Complex;

Complex complexAdd(Complex x, Complex y);

#include "complex.h"

Complex complexAdd(Complex x, Complex y) {
    Complex z;
    z.re = x.re + y.re;
    z.im = x.im + y.im;
    return z;
}
```
Example: Complex Numbers

- Structures may be initialized (the initial values are associated to the `struct` fields in order):
  
  ```
  Complex z = { 2.3, -.4 };
  ```

- Structures returned by functions can be used as such:
  
  ```
  double x = complexAdd(z1, z2).re;
  ```

- Assigning a structure copies all its fields. Structures are also passed to functions by copying (by value).
  
  ```
  z2 = z1; // copy re and im
  ```
typedef enum GenderType {
    MALE, FEMALE
} Gender;

typedef struct person {
    unsigned long id;
    char* name;
    Gender gender;
    struct { // unnamed struct
        char* street_address; // street & number
        char* city;
        unsigned int zip;
    } address; // a field of an unnamed struct
} Person; // abbreviation

#include "person.h"

unsigned long id, zip;
char *name1, *name2;
Person person, *person_ptr;

...

person.id = id;
(*person_ptr).id = ++id;
person_ptr->name = name1;
person.address.zip = person_ptr->address.zip;

 structs should generally come with **functions** for handling them
(as we will see next lecture)
A way to force an expression to be evaluated to a certain type. Consider:

```c
int i = 6;
double d = 2.9;
```

The following three expressions are evaluated to three different values:

```c
i/d ( == 2.0689)  (int)(i/d) ( == 2)  i/(int)d ( == 3)
```

Here we force an argument of a function to be of the correct type:

```c
d = sqrt((double)i);
```

Question: When is this casting computation executed (Compile time, link time, run time)?
int i, *pi; // pi is a pointer to an integer, *pi is an int

i = 3;
3: pi = &i; // now *pi == 3 by referencing i
4: *pi = 2; // now i == 2 by dereferencing pi

After line 3:

\[
\begin{array}{ccc}
\text{Address} & 0x6414 & 0x6480 \\
\hline
i & 3 & \text{pi} \\
\end{array}
\]

After line 4:

\[
\begin{array}{ccc}
\text{Address} & 0x6414 & 0x6480 \\
\hline
i & 2 & \text{pi} \\
\end{array}
\]
In C, function arguments are passed **by value**.

This means a variable used as an argument to a function **will always retain its value** when the function returns.

```c
void swap(int a, int b) {
    int t = a;
    a = b;
    b = t;
}
```

```c
int i = 3, j = 4;
swap(i, j);
// i is still 3 and j is still 4
```
Pointers and Function Arguments

- Pointers are used to overcome this restriction
  - Whenever we want a variable to be changed by a function, we must send its address (by value) to the function

```c
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

```c
int i = 3, j = 4;
swap(&i, &j);
// i is now 4 and j is now 3
```

- This makes the call to swap() practically a call by reference
Pointers and Function Arguments

- Pointers enable “returning” several values from a function
  - Useful when a function has to return one or more results, and also report whether it succeeded or failed

```c
typedef enum { FAIL, SUCCESS } Status;

Status readNumber(int* number) {
    int numRead = scanf("%d", &number);
    return numRead == 1 ? SUCCESS : FAIL;
}

Status wholeDivide(int a, int b, int* res, int* rem) {
    if (b == 0) return FAIL;
    *res = a / b;
    *rem = a - (*res) * b;
    return SUCCESS;
}
```

Equivalent to
```
return numRead == 1
```
but more readable
The status can also be returned using a pointer
- Allows us to return the actual computation result in the function’s return value

```c
Status array_max(int a[], int n, int* max);
```

```c
int a[5] = { ... }; int max;
array_max(a, 5, &max);
printf("maximum = %d", max);
```

fails if the array is empty

lots of **extra code** even though we know this cannot fail

if status is **not** NULL, writes the error to *status

```c
int array_max(int a[], int n, Status* status);
```

```c
int a[5] = { ... };
printf("maximum = %d", array_max(a, 5, NULL));
```

much better!
A pointer can have an integer added or subtracted from it:
- Moves the address according to the size of the type pointed to.
- Mostly used for arrays.

```
int i[N];  int* pi = &i[0];
```

So `pi + 1 = pi + 4` bytes (assuming `int` is 4 bytes...)

```
double d[N];  double* pd = &d[0];
```

So `pd + 1 = pd + 8` bytes (assuming `double` is 8 bytes...)
Through pointer arithmetic, a pointer can be used to access an entire array.

An operator [] for pointers simplifies this process even further, allowing them to be used essentially like arrays.

double weight[LEN];
double *pw = &weight[0];  // pw points to weight[0]
double x = pw[3];         // x gets weight[3]
pw += 2                   // pw points to weight[2]
pw[1] = x + 5;           // weight[3] is updated

ptr[k]

same thing

*(ptr + k)

pw = weight;
is also possible

after pw += 2
Pointers and Arrays

- Array names are implicitly cast to a pointer to their first element
  - However, this pointer is `const` and cannot be moved
- This cast does not happen within a `sizeof`, so the true array size is returned

```c
double weight[LEN];
double *pw = weight + 2; // weight is cast to the pointer &weight[0]
pw++;
weight++; // Error: weight is a constant

int n1 = sizeof(weight); // n1 is LEN*8 (assuming sizeof(double)==8)
int n2 = sizeof(pw); // n2 is 4 (or the system pointer size)
```
Arrays as Function Arguments

- When an array is passed to a function, its **address** is sent
  - The array is implicitly cast to a pointer to its first element

```c
void sort(int* v, int n) {
    ...
}

int x[1000];  // large array
sort(x, 1000);  // passes &x[0], and x gets sorted
sort(x+500, 500);  // sorts only the second half of the array
```

- A function that expects a pointer to the beginning of an array can emphasize this by using `[]` instead of `*` in its signature:
  - equivalent to writing “int* v”

```c
void sort(int v[], int n) {
    ...
}
```
We sometimes need to declare pointers without prior knowledge about the type they will point to.

The type `void*` is used as a **generic** pointer type.
- A `void*` pointer cannot be dereferenced (e.g., with `*` or `[]`) and cannot be used with pointer arithmetic, unless it is cast to its true type.

```c
int i = 1;
double d = 1.0;

void* ptr0 = &i;       // valid, but type is lost
void* ptr1 = &d;       // same here

double e = *ptr0 + *ptr1;    // compilation error
e = *(int*)ptr0 + *(double*)ptr1;  // the solution
```
**void*** Pointers

- A **void*** pointer can be used for generic **low-level memory operations**.
  
  avoid such code whenever possible

- **Tip:** type **char** is guaranteed to occupy exactly 1 **byte**.
  
  Thus, casting a **void*** to a **char*** allows accessing individual bytes in memory.

```c
void swap(void* p, void* q, size_t nbytes) {
    for (int i = 0; i < nbytes; i++) {
        char tmp = ((char*)p)[i];
        ((char*)p)[i] = ((char*)q)[i];
        ((char*)q)[i] = tmp;
    }
}
```

A typedef from `<stdlib.h>`. Defines the system-dependent type for counting bytes (usually int or unsigned int)

```c
double d1, d2;
int a[5], b[10];
...
swap(&d1, &d2, sizeof(double));
swap(a, b, 5*sizeof(int));
```
**Const and Pointers**

- A **constant pointer** cannot change **where** it points to

```c
int x = 5;
int* const ptr = &x; // ptr will always point to x
*ptr = 7; // fine, changes x
ptr++; // will not compile (ptr is const!)
```

- A **pointer to a const** cannot change **what** it points to

```c
int x = 5;
const int* ptr = &x; // ptr will never change the value of x
*ptr = 7; // will not compile (ptr points to a const)
ptr++; // fine, ptr points to another location
const int* const ptr2 = &x; // can combine both const types
int const* const ptr2 = &x; // is also fine
```
By taking a **pointer to const**, a function guarantees it will not modify the contents of that pointer:

```c
char* strcpy(char* str1, const char* str2); // copy str2 to str1
int strcmp(const char* str1, const char* str2); // compare str1 and str2
int strlen(const char* str); // length of str
```

A function taking a pointer which it does not intend to modify its content, **better** declare this using **const**

- Improves readability (distinguish between input and output arguments)
- Ensures the user's data remains intact
- End user responsibility (not enforced by the compiler)
- **Without this, pointers to constant data cannot be passed to the function:**

```c
char* bad_strcpy(char* str1, char* str2);
char str[100];
strcpy(str, "Source string"); // will compile
bad_strcpy(str, "Source string"); // warning or compilation error (string literal has type const char*)
```
C allows general purpose dynamic memory allocation on the heap, restricted only by the amount of memory available at run time.

There are four predefined standard functions for this:

- `void* malloc(size_t num_bytes);`
- `void* calloc(size_t num_elem, size_t elem_size);`
- `void* realloc(void* old_ptr, size_t new_size);`
- `void free(void* ptr);`

From `<stdlib.h>`

If a memory allocation fails, these functions return `NULL`.
- Note: `NULL` is not zero – it is represented as 0 in C (and C++) programs.

The only function that zeros the allocated bytes.
typedef struct {
    double re, im;
} Complex;

const int init_len = 10, new_len = 20;

Complex* vec1 = malloc(init_len * sizeof(Complex));
if (vec1 == NULL) {
    fprintf(stderr, "cannot allocate\n");
    exit(1);
}

Complex* vec2 = realloc(vec1, new_len * sizeof(Complex));
if (vec2 == NULL) {
    fprintf(stderr, "cannot allocate\n");
    exit(1);  // note the code duplication here
    // use an inline function or macro to prevent this
}

// vec1 now points "nowhere" - do not free
free(vec2); // return memory to heap - prevent memory leak

Why not use #define here?

what’s wrong with using exit?
The memory used by a process is divided logically to four areas:

- The **Stack** stores local (automatic) variables and calls to functions.
- The **Heap** stores data that is **dynamically** allocated by the program.
- The **Data** section stores global static variables and function static variables. Another part is marked as read-only and stores string constants.
- The **Code** section stores the machine code of the program (marked read only).
static double d1;
double d2, *pd;
static void PrintFunc1(const char *Str) {
    printf("%s\n", Str);
}
void PrintFunc2(const char *Str) {
    printf("%s\n", Str);
}

int SomeFunc(...) {
    char c, *s1 = &c, *s2;
    static char Str[10];
    {
        int i, *p = &i;
        ...
    }
    s1 = (char *) malloc(10);
    s2 = Str;
    pd = &d1;
    PrintFunc1(s1); PrintFunc2(s2);
}
static double d1;
double d2, *pd;

static void PrintFunc1(const char *Str) {
    printf("%s\n", Str);
}

void PrintFunc2(const char *Str) {
    printf("%s\n", Str);
}

int SomeFunc(...) {
    char c, *s1 = &c, *s2;
    static char Str[10];

    
    int i, *p = &i;
    ...

    s1 = (char *) malloc(10);
    s2 = Str;
    pd = &d1;
    PrintFunc1(s1); PrintFunc2(s2);
}

What is the difference between PrintFunc1 and PrintFunc2?
and between d1 and d2?

- **Stack (Automatic)** and **static data/variables** are managed automatically.
- **Heap (Dynamic)** data is not.
- Not freeing dynamically allocated data constitutes a memory leak.
Memory Management

void string_manipulation(char *str) { ...; free(str); }  
int main()
{
    char *str, *new_str, str2 = malloc(5);

    str = (char *) malloc(10);
    new_str = realloc(str, 20);
    string_manipulation(str);
    string_manipulation(new_str);
    printf("line %d: \tnew_str = %s\n", __LINE__, new_str);
}

Question: Can you identify problems in the above code?
Remember that all these (data) resources are limited!
What is wrong with the following code:

```c
void func1(...) {
    int Array2[10],
        *Array3 = (int *) malloc( sizeof(int) * 10);
    ... 
    func2(Array2);
    func2(Array3);
    ... 
}
```

```c
void func2(int *Arr) {
    int i;
    for (i = 0; i < 10; i++)
        *++Arr = i * i;
    for (i = 10; i-- >= 0; )
        Arr[i] += 2 * i;
}
```

- What can be the effect of such an under/overwrite error if the data is on the stack? On the heap?
- Detecting such errors can be quite challenging.
- See the “-fsanitize=address” option of gcc.

Any idea how?
"C is quirky, flawed, and an enormous success."

- Dennis M. Ritchie (1941-2011).