Chapter 1
The C Programming Language
(Advanced Features)
Some Simple Changes at C99

Improving Safety

• No implicit `int` (*)
• No implicit function declaration
• Strict `return` statements

Improving Expressiveness

• Compound literals (*)
• Hexadecimal floating-point constants
• Empty/variable-number macro arguments

(*) Look up terms you don’t know
Some Simple Changes at C99 (Cont.)

C++ Features that fit C Spirit

- // comments
- Inline functions
- Mixed declarations and code
- Declaring a variable in for-loop scope
- Boolean type bool in <stdbool.h>
- The type complex in <complex.h>
Some Simple Changes at C99 (Cont.)

Compensate for Developers Ignorance

- "%lf" is legal in printf
- Uniquely defined negative integral division

But beneficial for MACRO

Could Have Done Without It

- Variable length arrays
- long long int

Downgraded to optional in C11
C99: Inline functions

• The newer C standard (ANSI+ISO+IEC+9899-1999) allows inline functions (this feature was taken from C++)

• A function is inlined when its code is expanded at the point of the call to that function instead of creating a function-call.

• In that respect, inlining a function has the same effect as using a MACRO except one major difference

    Inlining is a job done by the compiler

• Consequently, it behaves like a function with all other respects, except code duplication and calling overhead

• The resulting executable may be smaller or larger.
Inline Functions (cont.)

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

- The function definition should be in a header (if it is used across several files)
- The function is restricted to a single set of types (as denoted in its definition)
  - Arguments of other types will be converted (if permissible)
  - If you need it for different types (without conversion), use MACRO
- Since the function must be defined before use it subverts top-down presentation
- The compiler may refuse to inline a function (either completely or in selected cases)
MACRO Definitions

#define identifier token-sequence
#define identifier( parameter-list ) token-sequence

• No space is allowed between the identifier and the parentheses.
• Every appearance of <identifier> in the code is replaced by the <token-sequence> prior to compilation. If arguments are involved, they replace the parameters.

• Common Practice:
  MACRO identifiers are all CAPITAL letters

• A Useful Rule: Constants present in software, with the possible exception of 0, 1, will backfire. USE MACROs.
MACRO examples – a rewrite will follow

```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))

char arr[MAX_STR_LEN + 1], *str;

for (str = arr; *str != '\0'; str++) {
    *str = TO_LOWER(*str);
}
```

 Erot 1. It assumes letters are ordered and continuous – true for ascii, not EBCDIC
 2. See the following slides for a safer style.
MACRO PIFALLS

Several problems may arise while calling the MACRO

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

- **Operator Precedence ERROR**
  - **External**
    ```c
    (int)SQR(a)
    (int)a * a
    ```
    is expanded to
    ```c
    (int)a * a
    ```
    which casts the left operand only
    solution: Put parentheses (or braces) around MACRO.
  - **Internal**
    ```c
    SQR(a + b)
    a + b * a + b
    ```
    is expanded to
    ```c
    a + b * a + b
    ```
    which is definitely
    ```c
    not (a + b)*(a + b)
    ```
    solution: Put parentheses around MACRO arguments.

```c
#define SQR(x) ((x) * (x))  /* is SAFE */
```
MACRO PITFALLS II

• **Side Effects ERROR**

  \[
  \text{SQR}(++i) \quad \text{is expanded to} \\
  ((++i) \times (++i)) \quad \text{which may increments } i \ \text{twice} \\
  \text{(and the result is } \text{undefined})
  \]

• **Unnecessary Function Calls** (actually, this is a *side-effect* too)

  \[
  \text{SQR(very_difficult_func(a,z,t))} \\
  \text{will evaluate the function twice, before multiplication.}
  \]

**NO General Solutions** for the above two problems:

Be *wise* while defining a MACRO

and *cautious* while calling a MACRO
MACRO examples – the right way

#define MAX_STR_LEN 20
#define IS_UPPER(c) ((c) >= 'A' && (c) <= 'Z') /* ascii 🙁 */
#define TO_LOWER(c) (IS_UPPER(c)? (c) - 'A' + 'a' : (c))

Note: the C preprocessor is a powerful tool, though, as we saw, not so safe. We will soon meet more of its capabilities, but at this stage you may want to learn about its # and ## operators that we are not going to teach.
Calling a **Function** vs. Calling a **MACRO**

* Is always an **expression**

* Will not change arguments and side-effects are fully controlled

* Can always return a newly created object

* Limited to **fixed type arguments**

* Saves executable code
  
  Easier maintenance

* **May** be passed as an argument to functions

* Function call overhead (for *stack* handling)

* May be a **statement** (e.g., one that requires *automatic* variables).

* May have unexpected side-effects

* May require an argument to carry a newly created object

* May operate unchangeably on arguments of varying types

* Code is **duplicated**

  though maintenance is easy

* **Cannot** be passed as an argument

* No calling overhead
When is a **MACRO** better than a **Function**?

**Rules of Thumb:**

- Operation required is **short**, **simple**, and (maybe) used in **different locations**. **Inlining may be better**

- Operation required is **short**, **simple**, and is used **intensively**. **Inlining may be better**

- Operation required is performed on variety of **different types**. **Inlining cannot do that**

**Examples for last case:**

```c
#define MAX(a, b)        ((a) > (b) ? (a) : (b))
#define SWAP(type, a, b) { type t = a; a = b; b = t;}
```
The following code compiles smoothly:

```c
if (strcmp(s, t) < 0) SWAP(char *, s, t);
if (x < VAL)
    x = MAX(y, z);
else
    SWAP(long double, x, y);
```

But the following code does not compile:

```c
if (strcmp(s, t) < 0) SWAP(char *, s, t);
if (x >= VAL)
    SWAP(long double, x, y);
else
    x = MAX(y, z);
```

Why?
General Solution:

Always use braces with \texttt{if-else} (good style)

\begin{verbatim}
if (strcmp(s,t) < 0) SWAP(char *, s, t);
if (x < VAL) {
    x = MAX(y,z);
} else {
    SWAP(long double, x, y);
}
\end{verbatim}

Using \texttt{do-while} there is a MACRO-trick that works around the problem

Even when it compiles without them

The wise doesn’t get into problems that the smart can solve
Enumerable Types

• **Types** that consist of certain **integral values**, which are carried by **symbolic names**. The names are more important than the actual values.

• **Enum definitions**
  ```c
  enum bool { FALSE, TRUE }; // C99 has pseudo bool
  enum month { JAN = 1, FEB, /* … */ , DEC };
  enum colors { WHITE = 1, BLACK, GREEN = 8, RED };
  ```

• **Using enum types**
  ```c
  enum bool b[SIZE], t = FALSE;
  ```

• **enum** vs. **#define** (enum is superior)
  – It obeys C’s scoping rules.
  – The compiler **may** check for type mismatch.
  – The debugger **may** recognize the symbolic names.
MACRO and Enum – a Comparison

```c
enum day {SUN = 1, MON, TUE, WED, THR, FRI, SAT};
void f(void)
{
    enum day d = TUE, /* o.k. */
    e = 13; /* Should have been a warning */
    enum bool {FALSE, TRUE};
    enum bool b = FALSE, /* o.k. */
    c = SAT; /* Should have been a warning */
#define RED 3
    int color = RED; /* o.k. */
}
void g(void)
{
    enum day d = TUE; /* o.k. */
    enum bool b = FALSE; /* syntax error */
    int color = RED; /* o.k. */
}
```

They are errors in C++
**const** type qualifier

```c
const int fixed = expression; /* No changes after initialization */
```

- When is `const` preferred over `enum` or a MACRO?
  - Its value is: of **any** type / decided at **run time** / **reevaluated** at initialization.
  - It obeys C’s scoping rules.
  - It is used where its address (& operator) is required.
  - It must be recognized by the compiler/debugger.
  - In trying to force a function not to modify an array argument, or any argument that is passed by its address.

**Example:** If a function is `declared` as
```c
int scalar_product(const int vec1[], const int vec2[]);
```
the compiler *may* check that no assignment of the form
```c
vec1[i] = expression;
```
is evaluated in the function body.
**const** and pointers

```c
const int fixed = expression ;
```
can also be written as

```c
int const fixed = expression ;
```
which is the recommended way

The benefits are visible for pointer declarations:

What is the difference between

```c
int const * ptr = &some_int;  and
int * const ptr = &some_int;  ?
```

```c
int const * const ptr = &some_int;  is also possible
```
**typedef Declarations**

C provides a facility for creating new data type **names**.

```c
typedef int Length;    /* Defining */
Length l, lvec[SIZE];  /* Using */
```

will make \(l\) of type **int**, and \(lvec\) of type array of int.

```c
typedef enum bool bool; /* Shortening */
```

- **Typedefs** are far from being MACROs:

```c
typedef char Buf[BUF_SIZ];
Buf buffer, buf_array[SIZE];
```

will make  \(buffer\) - an **array** (of size BUF_SIZ),
and \(buf_array\) - an **array** (of size SIZE)

of arrays (of size BUF_SIZ),

i.e, **equivalent to**

```c
char buf_array[SIZE][BUF_SIZ];
```
Why `typedef`?

• **Easy modification of data types**

   Example: Certain `int` variables are used for carrying flags. Later, the software became more complicated, and we want to change these variables into type `long`. Had these variables been declared being of type `Flag` (with "`typedef int Flag;`"), all can be done by modifying the `typedef` statement.

• **Meaningful names for data types**

   In the example above, wherever we see the declaration "`Flag var;`" we understand that ‘var’ is going to be used as a “flag carrier”.

• **It is a weak `typedef` (in C/C++)** – `Flag is int`. 
Casting

Is a way to force an expression to be evaluated to a certain type

Example:

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

– The following three expressions are evaluated to three different values:

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

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

```c
d = sqrt((double)i); /* a documentation benefit too */
```
Casting (Cont.)

- There are cases where we have to declare pointers without prior knowledge about the type they will point to.
- The type `void *` (i.e. a pointer to `void`) is used as a **generic** pointer type.
  - In a mixed type **pointer expression**, conversion is automatic.
  - However, **casting** is necessary when pointers are dereferenced.

**Example:**

```c
int i;
double d, e;
void *gpt0 = &i, *gpt1 = &d; /* Causes no problem */
e = *gpt0 + *gpt1; /* Is impossible */
e = *(int*)gpt0 + *(double*)gpt1; /* Is the solution */
```
Benefits of separate compilation?

A. None (E.g., Java and other PLs)
B. Shorter compilation (time)
C. Simpler compilation (complexity)
D. Independent compilation (textual)
E. Independent components (logical)
The C Preprocessor

- **MACRO definitions**
  
  ```c
  #define a macro definition
  #undef identifier
  ```

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

- **Conditional Compilation**
  
  ```c
  #if constant-expression
  #ifdef identifier /* #if defined(identifier) */
  #ifndef identifier /* #if !defined(identifier) */
  #elif constant-expression
  #else #endif
  
  #elif constant-expression
  #else
  #endif
  ```

Compile as

```
gcc -E a.c > a.E
```

In order to see it working
The C Preprocessor: Conditional Compilation

• In the header (usually)

```c
#ifdef __MS_WINDOWS__ /* Not the real flag */
    #include <conio.h>
    #define CLRSCR() clrscr()
#else
    #define CLRSCR() /* No need for UNIX */
#endif

• In the program

      ....
     CLRSCR(); /* Why was it defined with parentheses? */
    printf("A new calculation to solve:\n");
      ....
    CLRSCR();
```

A simple Example
A common use of the `#ifndef` command is in header files. It is, usually, harmful to include a header file more than once. Since file inclusion is transitive, a file may inadvertently be included more than once, through inclusion of other files. A strong mechanism that prevents multiple inclusion, is that each file defines a unique MACRO-identifier once included, and “refuses” inclusion if this identifier is `#defined`.

**Example:** *(a header file named “list.h”)*

```
#ifndef LIST_H
#define LIST_H    /* Prevents entering here in future inclusions */
(Here comes the content of the header file)
#endif    /* LIST_H */
```
Structures

Syntax:  \texttt{struct \ [name] \ {fields-list}}

- Define \textit{new types} that are built from several, simpler types.
- Allow access to each component.
- Almost everything that can be done with built-in types (like \texttt{int}, \texttt{double}), is legal for structures.
  
  Legal: Arrays of structures, functions that return structure values, etc.

  Illegal: Overloading predefined \textit{arithmetical / logical} operations,

  \textbf{Assignment between structures of the same type is legal.}

- Structures can be \textit{nested}.
A Simple struct: Complex Numbers

typedef struct complex complex;

struct complex {
    double r, i;
};

#define CMPLX_ADD(x,y,z) {(z).r = (x).r + (y).r; \
    (z).i = (x).i + (y).i; \
}

complex ComplexAdd(complex x, complex y) {
    complex z;
    z.r = x.r + y.r;
    z.i = x.i + y.i;
    return z;
}
Structures (Cont.)

Example: A nested struct

```c
struct person {
    unsigned long id;
    char *name;
    short gender;

    struct {
        char *str_n_num,
        *city;
        unsigned zip;
    } address;  /* a field of an unnamed struct */

    struct person *father;
    /* The above is legal because 'father' is only a pointer */
};
```

/* an unnamed struct */
**Structures (Cont.):**

**Usage:**

```c
unsigned long id, zip;
char *name1, *name2;
struct person per_var, *per_ptr;

per_var.id = id;
(*per_ptr).id = ++id;
per_ptr->name = name1;
per_var.address.zip = per_ptr->address.zip;
per_ptr->father->name = per_var.father->name;
```

- **Note:** The field operators (解放军, .) have the same precedence (the highest among all operators), and are associated `left to right.`
Structures (Cont.)

- Structures may be initialized:
  \[
  \text{complex } z = \{2.3, -0.4\};
  \]

- Structures returned by functions, may be used as such:
  \[
  \text{double } x = \text{ComplexAdd}(z_1, z_2).r;
  \]

- Structure declarations do not constitute executable code:
  They merely describe a template (a blueprint) to be used in building objects of a specific type. This blueprint describes the memory layout of these objects.
Addresses and Pointers
(Only for reference – make sure you understand it)

• Every object in the computer memory has an address.
• Some of the objects in a C program may be referenced through the address of their location in memory.
  - Expressions like &var, are evaluated to the address of var.
• The address operator, &, cannot be applied to objects that have a temporary location in the memory (examples are: explicit constants, compound expressions).
• Addresses can be stored in variables of type pointer to . . .
Addresses and Pointers  (Cont.)

• When `pvar` is a pointer variable carrying an address, the dereferencing (or indirection) operator, `*`, is used to extract the value stored in that address (via the expression `*pvar`).

• The character `*` is also used for the declaration of pointer type variables.
Addresses and Pointers (Cont.)

Example:

```c
int i, *pi;    /* pi - a pointer to integer */

/* Another words, *pi is an int */

i = 3; pi = &i;   /* Now, (*pi == 3) */

*pi = 2;    /* Now, (i == 2) */
```

Memory Image

<table>
<thead>
<tr>
<th>Address</th>
<th>0x6414</th>
<th>0x6480</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>3</td>
<td>0x6414</td>
</tr>
<tr>
<td>pi</td>
<td>0x6414</td>
<td></td>
</tr>
</tbody>
</table>

(After line 2, above)

<table>
<thead>
<tr>
<th>Address</th>
<th>0x6414</th>
<th>0x6480</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
<td>0x6414</td>
</tr>
<tr>
<td>pi</td>
<td>0x6414</td>
<td></td>
</tr>
</tbody>
</table>

(After line 3, above)
Addresses and Pointers (Cont.)

- In order to dereference a pointer, it must be known to which type it refers.
- Objects of different types may occupy spaces of different size, e.g., char, int, float, double.

Example: (Showing how pointers differ from integers.)

```c
char c[N]; ...
\[0 \quad 1\] ...
char *pc = &c[0]; (*(pc+1)==c[1]);

int i[N]; ...
\[0 \quad 1\] ...
int *pi = &i[0]; (*(pi+1)==i[1]);

double d[N]; ...
\[0 \quad 1\] ...
double *pd = &d[0]; (*(pd+1)==d[1]);
```

- It is illegal to compare two pointers, unless they are known to point to a single object (e.g., an array), or to be NULL. Illegal comparisons are many times possible, but the results may be surprising.
Pointers and Arrays

Following a declaration like

```
double weight[LEN], *pw;
```

we have:

- `weight[i]` is an expression of type `double` that refers to the value stored in the i’th entry of the array.
- `weight` is an expression of the type `LEN-size array of double`.
- Under **most** circumstances, the expression `weight` is evaluated **as if** it was of type `pointer to double` that refers to the address of the first element of the array.
  - This means that `weight == &weight[0]` is always **TRUE**.
  - However, `sizeof(weight)` will **not** evaluate to `sizeof(double*)`
- **Fact:** The C compiler always translates an “array expression” like `weight[i]` into its equivalent “pointer expression” *(weight+i)*
Pointers and Arrays (Cont.)

- However, when dealing with an object of type `array of ...` using the “array style expression” `weight[i]` is recommended, stressing the fact that we deal with an actual array.

- Assigning `pw = weight` will make the expression `pw[2]` have the value `weight[2]`.

- The main practical difference between `pw` and `weight` is that `weight` acts as a constant, and `pw` is a variable, i.e., `weight` cannot be assigned to.
Pointers and Arrays (Cont.)

Example:

```c
pw = weight + 2;
```

```
weight [ ] [ ] [ ] [ ] [ ] [ ] [ ]...

pw [ ]
```

```
pw++;  (now pw[1] is weight[3])
```

```
weight [ ] [ ] [ ] [ ] [ ] [ ] [ ]...

pw [ ]
```

**BUT** weight++;  *is illegal.*
Points and Function-Arguments

• In the C language, function arguments are passed only by value.
• It means that a variable that is used as an argument at a function call, will always retain its value when the function returns to the caller.

Example:

```c
int i = 3, j = 4;
swap(i, j);
/* i is still 3,
 * j is still 4.
 */

void swap(int a, int b)
{
    int t = a;
    a = b;  b = t;
    return;
}
```
Pointers and Function-Arguments
(Cont.)

• By using pointers as function arguments this restriction is overcome.

Example:

```c
int i = 3, j = 4;
swap(&i,&j);
/*  i is now  4,           void swap(int *a, int *b)
 *  j is now  3.            {  int t = *a;
 */  *a = *b;  *b = t;
 return;
} 
```

• This makes the call to `swap()`, practically a call by reference.

• In both cases, `a`, `b`, are local variables, which are initialized to the values of the arguments at the function call.
Arrays as Function-Arguments

(End of “Addresses and Pointers” reference)

An exception to the C “call by value” paradigm

is for array parameters

• When an array is used as an argument at a function call, the entire array is not copied, but only its address is passed.

  – For a one-dimensional array of integers vec[SZ], that is used as an argument at a function call, \( f(\text{vec}) \) and \( f(&\text{vec}[0]) \) are synonyms.
  – In that case, at the definition or declaration of the function \( f \), \( f(\text{int } *\text{arr}) \) and \( f(\text{int } \text{arr}[]) \) are synonyms too.
  – If part of the array is to be transferred at the function call, then, e.g., \( f(\text{vec}+2) \) and \( f(&\text{vec}[2]) \) are synonyms.
  – For multidimensional arrays, the above discussion is true for the first (leftmost) coordinate only.
Pointers to Functions

There are cases where there is a function call in a command, but there is no prior knowledge which function is to be called.

Example:

```c
void *v1, *v2;
...
if (compare(v1, v2) == 0) {
  ...
}
```

`v1, v2` may point to integers or strings, or other types. An appropriate `compare` function should be called, according to the type of the objects pointed to by `v1, v2`.

The function `compare()` is useless:
1. It requires additional “type” argument.
2. `compare()` will require modification every time a new type is added.
Pointers to Functions (Cont.)

This is how we deal with the situation:

```c
void *v1, *v2;
enum {INT, STR} type;   /* coding actual type */
int (*compare)(void*,void*); /* a pointer to function */

switch(type) {  
    case INT:  compare = int_cmpr;
                break;

    case STR:  compare = str_cmpr; /* calls strcmp() */
                break;
}

if (*((compare)(v1,v2) == 0) { /* compare(v1,v2) is legal too */

    . . .
```

```c
int int_cmpr(void *p,void *q)
{
    return *(int*)p - *(int*)q;
}
```

Click Click Click Click
This is how we deal with the situation:

```c
void *v1, *v2;
enum {INT, STR} type;   /* coding actual type */
int (*compare)(void*,void*); /* a pointer to function */

switch(type) {  
    case INT:  compare = int_cmpr;
                break;

    case STR:  compare = str_cmpr; /* calls strcmp() */
                break;
}

if (*((compare)(v1,v2) == 0) { /* compare(v1,v2) is legal too */

    . . .
```
Another case of using a pointer to function is when a function is used as an argument, passed to another function (in principle, this is similar to the previous case: had we known a-priori a single function that would be used, there was no need to put it as an argument).

Example:

- A definition of a function which uses a function as an argument:

```c
char *
string_manipulation(char s[], int (*chr_manip)(int))
{
    for (int i = 0; i < strlen(s); i++)
        s[i] = chr_manip(s[i]);    /* A given operation on a char */
    return s;
}
```

- A call to that function:

```c
char str[10] = "aBcD";

...  

string_manipulation(str, tolower);
```
Dynamic Memory Allocation

• C allows general purpose dynamic memory allocation off the heap, restricted only by the amount of memory available at run time.

• There are three predefined functions for this: (in <stdlib.h>)

```c
void *ptr = malloc(num_bytes_to_allocate);
void *z_ptr = calloc(num_of_obj, size_of_1_obj);
void *new_ptr = realloc(old_ptr, new_size_in_bytes);
```

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

• C99 provides variable-length arrays, but it does not eliminate the need for dynamic memory allocation. Example of a VLA:

```c
void f(int n) { double vec[n]; . . . }
```
Dynamic Memory Allocation
(Cont.)

Example:
```c
int *vec, *new;

if ((vec = (int*)malloc(arr_lng*sizeof(int))) == NULL) {
    fprintf(stderr,"cannot allocate\n");
    exit(1);  /* a MACRO will save you from code duplication */
}
if ((new = (int*)realloc(vec,new_lng*sizeof(int))) == NULL) {
    fprintf(stderr,"cannot allocate\n");
    exit(1);
}  /* Now, vec points “nowhere” */
```

- **Dynamically allocated** memory (only) can be returned to the system using the function `free(old_ptr)`, which returns **void**.
Dynamic Memory Allocation

Example

```c
#include <stdio.h>
#include <stdlib.h>  /* For malloc(), etc. */
#include <ctype.h>   /* For tolower(), etc. */
#define LNG1 (6)
#define LNG2 (7)

char*
string_manipulation(char *s, int (*chr_manip)(int))
{
    while (*s != '\0') {
        *s = chr_manip(*s);
        s++;
    }
    return s;
}
```
Dynamic Memory Allocation

Example (Cont.)

```c
int main()
{
    char *str, *new_str, *str1;

    str = (char *) malloc(LNG2);

    while (printf("Enter 6-char string\n")
        && fgets(str,LNG2,stdin)) {
        printf(" %s ====> ",str);
        string_manipulation(str, tolower);
        printf("%s\n",str);
    }
```
new_str = realloc(str, LNG2+LNG1);
printf("line %d:\tnew_str = %s\n", __LINE__, new_str); (line 33)
printf("line %d:\tstr = %s\n", __LINE__, str); (line 34)
string_manipulation(str, toupper);
printf("line %d:\tnew_str = %s\n", __LINE__, new_str); (line 36)
printf("line %d:\tstr = %s\n", __LINE__, str); (line 37)

str1 = malloc(LNG2);
printf("line %d:\tstr1 = %s\n", __LINE__, str1); (line 40)
string_manipulation(str, tolower);
printf("line %d:\tstr1 = %s\n", __LINE__, str1); (line 42)
return 0;
}
Dynamic Memory Allocation

Example (Cont.)

After line

31
str
new_str
str1

32
str
new_str
str1

35
str
new_str
str1

41
str
new_str
str1
Dynamic Memory Allocation
Example (Cont.)

A Sample Output

Enter 6-char string
   CDfsER ==>> cdfser
Enter 6-char string
   ZYXABC ==>> zyxabc
Enter 6-char string
   line 33: new_str = zyxabc
   line 34: str = zyxabc
   line 36: new_str = zyxabc
   line 37: str = ZYXABC  This is a FREE memory area !!!
   line 40: str1 = ZYXABC  A FREE memory is manipulated
   line 42: str1 = zyxabc  This area is ALLOCATED again
                                 and manipulated by an OLD pointer
void printBestStudent(const char *names[], int n, double *gr[], int gr_size[])
{
    int i, j;
    double student_sum = 0.0, best_avg, curr_avg;
    const char *best_name;
    if (n <= 0) return;
    for (i = 0; i < gr_size[0]; i++) {
        student_sum += gr[0][i];
    }
    best_avg = student_sum / gr_size[0];
    best_name = names[0];
    for (i = 1; i < n; i++) {
        student_sum = 0.0;
        for (j = 0; j < gr_size[i]; j++) {
            student_sum += gr[i][j];
        }
        curr_avg = student_sum / gr_size[i];
        if (curr_avg > best_avg) {
            best_name = names[i];
            best_avg = curr_avg;
        }
    }
    printf("Best student is %s", best_name);
}
void printBestStudent(Student student[], int n)
{
    int i, j;
    double student_sum = 0.0, best_avg, curr_avg;
    Student best_student;
    if (n <= 0) return;
    for (i = 0; i < student[0]->num_grades; i++) {
            student_sum += student[0]->grade[i];
    }
    best_avg = student_sum / student[0]->num_grades;
    best_student = student[0];
    for (i = 1; i < n; i++) {
            student_sum = 0.0;
            for (j = 0; j < student[i]->num_grades; j++) {
                    student_sum += student[i]->grade[j];
            }
            curr_avg = student_sum / student[i]->num_grades;
            if (curr_avg > best_avg) {
                    best_student = student[i];
                    best_avg = curr_avg;
            }
    }
    printf("Best student is %s", best_student->name);
}
void printBestStudent(Student student[], int n) {
    if (n <= 0) return;
    double student_sum = 0.0;
    for (int i = 0; i < student[0]->num_grades; i++) {
        student_sum += student[0]->grade[i];
    }
    double best_avg = student_sum / student[0]->num_grades;
    Student best_stud = student[0];

    for (int i = 1; i < n; i++) {
        double student_sum = 0.0;
        for (int j = 0; j < student[i]->num_grades; j++) {
            student_sum += student[i]->grade[j];
        }
        double curr_avg = student_sum / student[i]->num_grades;
        if (curr_avg > best_avg) {
            best_stud = student[i];
            best_avg = curr_avg;
        }
    }
    printf("Best student is %s", best_student->name);
}
```c
void printBestStudent(Student student[], int n)
{
    if (n <= 0) return;
    double student_sum = 0.0;
    for (int i = 0; i < student[0]->num_grades; i++) {
        student_sum += student[0]->grade[i];
    }
    double best_avg = student_sum / student[0]->num_grades;
    Student best_stud = student[0];

    for (int i = 1; i < n; i++) {
        double student_sum = 0.0;
        for (int j = 0; j < student[i]->num_grades; j++) {
            student_sum += student[i]->grade[j];
        }
        double curr_avg = student_sum / student[i]->num_grades;
        if (curr_avg > best_avg) {
            best_stud = student[i];
            best_avg = curr_avg;
        }
    }
    printf("Best student is %s", best_stud->name);
}
```
double studentAverage(Student student)
{
    double student_sum = 0.0;
    for (int i = 0; i < student->num_grades; i++) {
        student_sum += student->grade[i];
    }
    return student_sum / student->num_grades;
}

void printBestStudent(Student student[], int n)
{
    if (n <= 0) return;

    Student best_stud = student[0];
    double best_avg = studentAverage(best_stud);

    for (int i = 1; i < n; i++) {
        double curr_avg = studentAverage(student[i]);
        if (curr_avg > best_avg) {
            best_student = student[i];
            best_avg = curr_avg;
        }
    }
    printf("Best student is %s", best_student->name);
}
Separation of Concerns

**Compare**

```cpp
Student bestStudent(Student student[], int n)
{
    Student best_stud = student[0];
    double best_avrg = studentAverage(best_stud);
    for (int i = 1; i < n; i++) {
        double const curr_avg = studentAverage(student[i]);
        if (curr_avg > best_avrg) {
            best_stud = student[i];
            best_avrg = curr_avg;
        }
    }
    return best_stud;
}
```

**Precondition:** $n > 0$

**Postcondition:** $best\_stud \neq NULL$

**Evaluate**

```cpp
double studentAverage(Student student)
{
    double student_sum = 0.0;
    for (int i = 0; i < student->num_grades; i++) {
        student_sum += student->grade[i];
    }
    return student_sum / student->num_grades;
}
```

**Print**

```cpp
void printBestStudent(Student student[], int n)
{
    if (n <= 0) return;
    printf("Best student is %s", bestStudent(student, n)->name);
}
```
A Simple Example

PlumberStatus
open_tap (const string tap_name)
{
    LockSys_Mutex LL(tap_lock_);
    const_iterator it = taps_find(tap_name);
    if (it == taps_end()) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(true);
    return PLUMB_OK;
}
What’s The Difference?

PlumberStatus
close_tap(const string tap_name)
{
    LockSys_Mutex LL(tap_lock_);
    const_iterator it = taps_find(tap_name);
    if (it == taps_end()) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(false);
    return PLUMB_OK;
}
A Simple Example

PlumberStatus

open_tap (const string tap_name)
{
    LockSys_Mutex LL(tap_lock_);
    const_iterator it = taps_find(tap_name);
    if (it == taps_end()) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(true);
    return PLUMB_OK;
}
What’s The Difference?

PlumberStatus
close_tap(const string tap_name)
{
    LockSys_Mutex LL(tap_lock_);
    const_iterator it = taps_find(tap_name);
    if (it == taps_end()) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(false);
    return PLUMB_OK;
}
What’s The Problem?

- This is a real code from industry
  “Modified” from C++ (and nouns renamed to hide context)
- Is it code duplication?
  – After extracting out the common parts we get

```cpp
PlumberStatus
Tap::open_tap(const string& tap_name) {
    LockSys<Mutex> LL(tap_lock_);
    if (!tap_found(tap_name)) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(true);
    return PLUMB_OK;
}
```
Code Duplication is just the Symptom

The real problem: Each one of the functions has two tasks

- Delegation
- Wrapping: a boolean value => function-name
Single Task Implementation: Delegation

PlumberStatus
operate_tap(const string name, bool open)
{
    LockSys_Mutex LL(tap_lock_);
    const_iterator it = taps_find(tap_name);
    if (it == taps_end()) {
        return PLUMB_TAP_NOT_FOUND;
    }
    it->second->operate(open);
    return PLUMB_OK;
}
Single Task Implementation: Wrappers

PlumberStatus
open_tap(const string tap_name)
{ return operate_tap(tap_name, true);} 

PlumberStatus
close_tap(const string tap_name)
{ return operate_tap(tap_name, false);}
The Original has a Third Problem

It enforces awkward usage

```cpp
if (activation_required) {
    open_tap(name);
} else {
    close_tap(name);
}
```

Instead of

```cpp
operate_tap(name, activation_required);
```
The Biggest Misconception About Functions

The Purpose of Functions is to Eliminate Code Duplication

The Purpose of Functions is to make the Code Easier to Understand

- By naming a piece of code (saving comments)
- By hiding its implementation (high level code)
- By making pre/post-conditions explicit
  - Also allowing (partial) isolation for testing
- By making the hosting code/function shorter