CH-230-A

Programming in C and C++

C/C++

Tutorial 5

Dr. Kinga Lipskoch

Fall 2019

The `const` Keyword

- The modifier `const` can be applied to variable declarations
- It states that the variable cannot be changed
  - i.e., it is not a variable but a constant
- When applied to arrays it means that the elements cannot be changed

const Examples

```
const double e = 2.71828182845905;
const char str[]= "Hello world";
e = 3;              /* error */
str[0] = 'h';       /* error */
```

- You can also use #define of the preprocessor
- But defines do not have type checking, while constants do

More `const` Examples

- `const char *text = "Hello";`
  - Does not mean that the variable `text` is constant
  - The data pointed to by `text` is a constant
  - While the data cannot be changed, the pointer can be changed
- `char *const name = "Test";`
  - `name` is a constant pointer
  - While the pointer is constant, the data the pointer points to may be changed
- `const char *const title = "Title";`
  - Neither the pointer nor the data may be changed

Dealing with Big Projects

- Functions are a first step to break big programs in small logical units
- A further step consists in breaking the source into many files
  - Smaller files are easy to handle
  - Objects sharing a context can be put together and easily reused
- C allows to put together separately compiled files to have one executable

Declarations and Definitions

- **Declaration**: introduces an object. After declaration the object can be used
  - Example: functions' prototypes
- **Definition**: specifies the structure of an object
  - Example: function definition
- Declarations can appear many times, definitions just once

Building from Multiple Sources

- C compilers can compile multiple sources files into one executable
- For every declaration there must be one definition in one of the compiled files
  - Indeed also libraries play a role
  - This control is performed by the linker
- `gcc -o name file1.c file2.c file3.c`

Libraries

- Libraries are collection of compiled definitions
- You include header files to get the declarations of objects in libraries
- At linking time libraries are searched for unresolved declarations
- Some libraries are included by `gcc` even if you do not specifically ask for them

Linking Math Functions: Example

```
#include <math.h>
#include <stdio.h>

int main() {
  double n;
  double  sn;

  scanf("%lf\n", &n); /* double needs %lf */
  sn = sqrt(n);
  /* conversion from double to float ok */
  printf("Square root of %f is %f\n", n, sn);
  return 0;
}

    gcc -lm -o compute compute.c
```

Compilers, Linkers and More

- Different compilers differ in many details
  - Libraries names, ways to link against them, types of linking
- Check your documentation
- But preprocessing, compilation and linking are common steps

Recursive Functions (1)

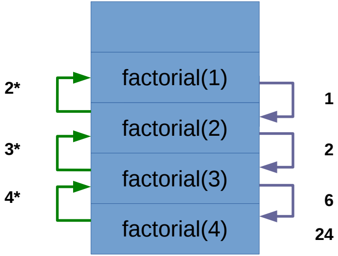
- Can a function call other functions?
  - Yes, indeed function calls appear only inside other functions (and everything starts with the execution of `main`)
- Can a function call itself?
  - Yes, but in this case special care should be taken
- A function which calls itself is called a **recursive function**
- Function A calls function A
- At a certain point function B calls A
  - A calls A then A calls A then A calls A ...
- When coding recursive functions attention should be paid to avoid endless recursive calls

Recursive Functions (2)

- Recursion theory can be studied for a longer time: here we will just scratch its surface from a basic coding standpoint
- Every recursive function must contain some code which allows it to terminate without entering the recursive step
  - Usually called **inductive base** or **base case**
- When recursion is executed, the new call should be driven "towards the inductive case"

Stack of Calls: Example

```c
int factorial(int n) {
  if ((n == 0) || (n == 1))
    return 1;
  else
    return n * factorial(n - 1);
}
```



Tracing the Stack of Calls (1)

```c
int factorial(int n) {
  int val;
  if ((n == 0) || (n == 1)) {
    printf("base\n");
    return 1;
  } else  {
    printf("called with par = %d\n", n);
    val = n * factorial(n - 1);
    printf("returning %d\n", val);
    return val;
  }
}
int main() {
  printf("%d\n", factorial(4));
  return 0;
}
```

Tracing the Stack of Calls (2)

From the `main`: call `factorial(4)`

factorial(1): n = 1, printf("base"), return 1

factorial(2): n = 2, printf(2), val = 2 * factorial(1), printf(val), return val

factorial(3): n = 3, printf(3), val = 3 * factorial(2), printf(val), return val

factorial(4): n = 4, printf(4), val = 4 * factorial(3), printf(val), return val

~~factorial(1): n = 1, printf("base"), return 1~~

~~factorial(2): n = 2, printf(2), val = 2 * 1, printf(2), return 2~~

~~factorial(3): n = 3, printf(3), val = 3 * 2, printf(6), return 6~~

~~factorial(4): n = 4, printf(4), val = 4 * 6, printf(24), return 24~~

One More Example: Fibonacci Numbers

$$F(N) = \begin{cases} 1, & N \leq 1 \\ F(N-1) + F(N-2), & N > 1 \end{cases}$$

```
int fibonacci(int n) {
  if ((n == 0) || (n == 1))
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

The C Preprocessor (1)

- Before compilation, C source files are being preprocessed
- The preprocessor replaces tokens by an arbitrary number of characters
- Offers possibility of:
  - Use of named constants
  - Include files
  - Conditional compilation
  - Use of macros with arguments

The C Preprocessor (2)

- The preprocessor has a different syntax from C
- All preprocessor commands start with `#`
- A preprocessor directive terminates at the end-of-line
  - Do not put `;` at the end of a directive
- It is a common programming practice to use all uppercase letters for macro names

The C Preprocessor: File Inclusion

- `#include <filename>`
  - includes file, follows implementation defined rule where to look for file, for Unix is typically `/usr/include`
  - Ex: `#include <stdio.h>`
- `#include "filename"`
  - looks in the directory of the source file
  - Ex: `#include "myheader.h"`
- Included files may include further files
- Typically used to include prototype declarations

The C Preprocessor: Motivation for Macros (1)

- Motivation for using named constants/macros
- What if the size of arrays has to be changed?

```
int data[10];
int twice[10];
int main()
{
  int index;
  for(index = 0; index < 10; ++index) {
    data[index] = index;
    twice[index] = index * 2;
  }
  return 0;
}
```

The C Preprocessor: Motivation for Macros (2)

More generic program if using named constants/macros

```
#define SIZE 20
int data[SIZE];
int twice[SIZE];
int main()
{
  int index;
  for(index = 0; index < SIZE; ++index) {
    data[index] = index;
    twice[index] = index * 2;
  }
  return 0;
}
```

Works but it no type information is associated with macros, so using `const` for this problem is a better solution.

The C Preprocessor: Macro Substitution (1)

- Definition of macro
  - `#define NAME replacement_text`
- Any name may be replaced with any replacement text
  - Ex: `#define FOREVER for (;;)` defines new word FOREVER to be an infinite loop
  - Ex:

```
#define ODD(A, B) { unsigned char abit=A & 1;\
                    unsigned char bbit=B & 1; \
                    ... }
```

The C Preprocessor: Macro Substitution (2)

- Possible to define macros with arguments
  - `#define MAX(A, B) ((A) > (B) ? (A) : (B))`
- Each formal parameter (A or B) will be replaced by corresponding argument
  - `x = MAX(p+q, r+s);` will be replaced by
  - `x = ((p+q) > (r+s) ? (p+q) : (r+s));`
- It is type independent

The C Preprocessor: Macro Substitution (3)

- Why are the `( )` around the variables important in the macro definition?
  - `#define SQR(A) (A)*(A)`
- Write a small program using this and see the effect without `( )` in `(A)*(A)` by calling `SQR(5+1)`
- Try also `gcc -E program.c` sends the output of the preprocessor to the standard output
- What happens if you call `SQR(++i)`?

The C Preprocessor: Macro Substitution (4)

- Spacing and syntax in macro definition is very important
- See the preprocessor output of the following source code

```
#include<stdio.h>
#define MAX =10
int main()
{
  int counter;
  for(counter =MAX; counter > 0; --counter)
    printf("Hi there!\n");
  return 0;
}
```

wrong_macro.c