MU5EEH11
C Programming
A small introduction

Ludovic Saint-Bauzel
ludovic.saint-bauzel@sorbonne-universite.fr

Sorbonne Université
Institut des Systèmes Intelligents et de Robotique (CNRS UMR 7222)

2023-2024

Course Information

This course is based on the C programming language, which is the base of UNIX operating system and embedded processors and micro-controllers. After an introduction to the C language, the course focuses on advanced concepts, such as dynamic memory allocation, processes, inter-process communication.

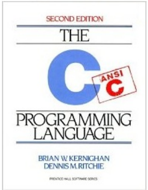
Contact:

- Ludovic Saint-Bauzel, 55-56/100

✉ : `ludovic.saint-bauzel@sorbonne-universite.fr`

References

B.W. Kernighan & D.M. Ritchie. **The C Programming Language**. 2nd ed. Prentice Hall, 1988.



C programming language

History

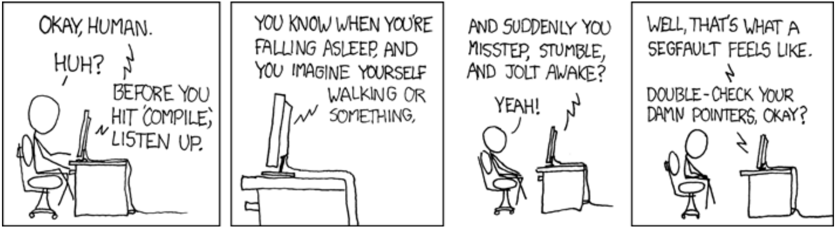
- 1972: Dennis Ritchie - AT&T Bell Labs
- 1978: "The C programming language" published
- 1989: C89 standard, aka ANSI C or standard C
- 1990: C90 is adopted by ISO
- 1999: C99 standard
- C11 (2011), C17 (2018 ; current latest standard), C2x (in development expected 2023)

Widely used

- OS, like Linux
- μcontrollers
- embedded processors
- DSP processors

Features

- Pro: few keywords, structures, pointers (memory, arrays, ..),external libraries, faster code
- Cons: no exceptions, no object-oriented programming, no polymorphism



(original comics on xkcd.com)

Beware! C is inherently unsafe!

- Low-level language
- No exceptions
- No range checking
- No type checking at runtime

Writing C code

Editing: hello.c

```c
#include <stdio.h> // printf
int main(){
    printf("Hello␣World!\n");
  return 0;
}
```

Compiling:

```
gcc -Wall hello.c -o hello
```

Includes and prototypes

Directives #include leads to the inclusion of *headers*) containing the **prototypes** of functions
Exemple : dans #include <stdio.h> :

- Ecriture formatée : int printf(const char *format, ...);

Dans #include <stdlib.h> :

- Conversion d'une chaîne en entier : int atoi(const char *nptr);
- Conversion d'une chaîne en double : double atof(const char *nptr);

man

Pour connaître dans quel header chercher une fonction, on peut utiliser le manuel (man). Par exemple (dans le shell) : man atof

```
ATOF(3)                 Manuel du programmeur Linux                 ATOF(3)

NOM
       atof - Convertir une chaîne en réel double précision (double)

SYNOPSIS
       #include <stdlib.h>

       double atof(const char *nptr);

DESCRIPTION
       La  fonction atof() convertit le début de la chaîne pointée par nptr en
```

Le manuel

- `man` est très utile. Il a aussi un manuel (`man man`)
- Pages divisées en 9 sections, les plus utiles :
  - 1 : commandes shell standard (ex: `ls`)
  - 8 : commandes shell admin (ex: `adduser`)
  - 3 : fonctions C fournies par les bibliothèques (ex: `atof`)
  - 2 : fonctions C correspondant à des appels système (fournies par l'API du noyau; ex: `read`)
- Pour chercher une page dans une section particulière, donner son numéro avant le nom (ex: `man 3 printf`) Sinon, renvoie l'entrée dans la première section où elle est trouvée.
- Pour les fonctions C, donne (en général) :
  - Description, prototype, header correspondant
  - Fonctionnement, valeur de retour, arguments
  - Gestion d'erreurs
  - Parfois, exemples de code
- Qualité et quantité très variables (de très sommaire à beaucoup trop détaillé !)

↦ **Prototype générique de la fonction main :** `int main(int argc, char* argv[]);`

- La fonction main d'un programme peut prendre des arguments en ligne de commande.
- Exemple : » `cp fichier1.tex fichier2.tex`
- La récupération de ces arguments se fait au moyen des arguments `argc` et `argv`.
- `argc` : nombre d'arguments passés au main $+ 1$
- `argv` : tableau de pointeurs sur des chaînes de caractères :
  - le premier élément `argv[0]` pointe sur la chaine qui contient le nom du fichier exécutable du programme.
  - les éléments suivants, `argv[1]`, `argv[2]`, ..., `argv[argc-1]` pointent sur les chaines correspondants aux arguments passés en ligne de commande.

↪ **Les arguments du main offrent de nombreuses possibilités.**

TEST 1

Given test.c

```
/* a test */
#include<stdio.h>
int main(int argc, char* argv[])
{
puts("Hello students!"); //output text to stdout and end line
return 0;
}
```

and the following definition:

```
#define DMSG "Hello students!"
```

How do we rewrite test.c

```
/* a test using define */
#define DMSG "Hello students!"
#include<stdio.h>
int main(int argc, char* argv[])
{
 const char msg[] = DMSG;
 puts(msg);
 return 0;
}
```

TEST 2

```
#define DMSG = "Hello students!"
```

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

```
int function (void arg)
{ return arg-1; }
```

Find the error in the statements above, and fix it:

TEST 2

```
#define DMSG = "Hello students!"
```

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

```
int function (void arg)
{ return arg-1; }
```

Find the error in the statements above, and fix it:

```
#define DMSG "Hello students!"
```

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

```
int function (int arg)
{ return arg-1; }
```

Plan

- `type name;`
- Declaration associate the variable *name* to its `type`. The compiler reserve the memory space based on this information in the *stack* of the program.

type name;

Data Types in C

- `char` : character
- `int` : integer
- `float` : floating point number
- `double` : double precision floating point number
- `void` : no type
- `size_t` : size of an object

hello_types.c

```
#include <stdio.h> // printf
int main() {
  char letter = 'a';
  printf("letter : %c size : %lu\n", letter, sizeof(letter));
  int decimal = 42;
  printf("decimal %d size : %lu\n", decimal, sizeof(decimal));
  float f1 = 1230.232;
  float f2 = -42.10;
  printf("f1 : %f \n f2: %f size : %lu\n", f1, f2, sizeof(f1));

  return 0;
}
```

...

hello_types.c

```
#include <stdio.h> // printf
int main() {
  char letter = 'a';
  printf("letter : %c size : %lu\n", letter, sizeof(letter));
  int decimal = 42;
  printf("decimal %d size : %lu\n", decimal, sizeof(decimal));
  float f1 = 1230.232;
  float f2 = -42.10;
  printf("f1 : %f \n f2: %f size : %lu\n", f1, f2, sizeof(f1));

  return 0;
}
```

```
letter : a size : 1
decimal 42 size : 4
f1 : 1230.232056
f2: -42.099998 size : 4
```

Plan

- Arrays are a collection of elements of the same data type.
- They provide a way to store multiple values under a single name.
- Array elements are accessed by their index.

Declaring and Initializing Arrays

```c
int numbers[5];              // Declaration
int data[] = {1, 2, 3};      // Declaration with initiali
char name[10] = "John";       // Character array (string)
```

- Arrays can be declared and initialized in various ways.
- Strings in C are essentially character arrays.

Strings in C

- In C, strings are arrays of characters terminated by a null character (`'\0'`).
- String manipulation functions in C use null-terminated strings.

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

int main() {
        char greeting[6] = "Hello";
        printf("Greeting: %s\n", greeting);
        strcat(greeting, " World");
        printf("Modified Greeting: %s\n", greeting);
        return 0;
}
```

- The `strcat()` function appends one string to another.

Memory Allocation with `malloc()`

- `malloc()` is used to dynamically allocate memory at runtime.
- It returns a pointer to the allocated memory.
- `malloc()` is part of the `stdlib.h` library.

Using malloc()

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

int main() {
        int *numbers;
        numbers = (int *)malloc(5 * sizeof(int));
        if (numbers != NULL) {
                numbers[0] = 1;
                numbers[1] = 2;
                // ...
                free(numbers); // Release

        }
        return 0;
}
```

- `malloc()` is used to allocate memory for an array of integers.
- `free()` is used to release the allocated memory to prevent memory leaks.

Pointers in C

- Pointers are variables that store memory addresses.
- They are used to access and manipulate data indirectly.
- Pointers are an essential concept for dynamic memory allocation and efficient data handling in C.

Using Pointers

```c
#include <stdio.h>

int main() {
        int num = 42;
        int *ptr = &num; // Pointer to an integer
        printf("Value of num: %d\n", num);
        printf("Value using pointer: %d\n", *ptr);
        return 0;
}
```

- The * operator is used to declare a pointer and access the value it points to.
- The & operator is used to get the memory address of a variable.

Summary: Arrays, Strings, `malloc()`, and Pointers

- Arrays are collections of elements of the same type.
- Strings are null-terminated character arrays used for text manipulation.
- `malloc()` dynamically allocates memory at runtime.
- Pointers store memory addresses and are crucial for dynamic memory management.

Plan

Plan

↦ About Memory Allocation...

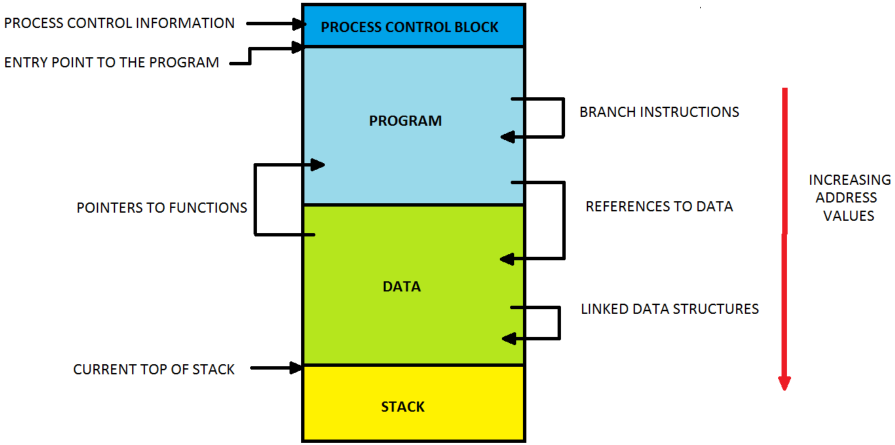
Different Allocation Methods

- **Static** allocation, at program launch; This applies to global variables.
- **Automatic** allocation, in the execution stack of the corresponding process for the running program; This applies to local variables in functions with a known and specified size at the time of program writing; It also includes memory allocated for passing arguments to a function.
- **Dynamic** allocation, in the heap, which is a memory area of the system not specifically dedicated to the currently running program.

Some Remarks

- Dynamic allocation provides great programming flexibility because it doesn't require the allocation of arbitrarily large chunks of memory at program launch.
- However, it can be relatively time-consuming, may sometimes fail (difficult to recover from), and can be a source of many bugs.
- Whenever possible, it's preferable to avoid dynamic allocation instructions in critical parts of the program.
- For example, dynamically allocating memory in the part of the code that performs high-frequency control loops should be avoided.

↦ **a process in memory**

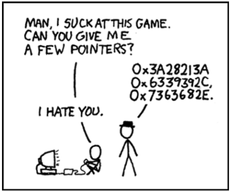


Plan

(original comics on xkcd.com)

pointers are **memory addresses**

symbol	example	meaning
`&`	`&x`	the address of `x`
`*`	`*p`	the object pointed by `p`



`int *a, *b;`	declare int pointers `a,b`
`int c;`	declare int `c`
`a = &c;`	`a` has the address of `c`
`*b = *a;`	the object pointed by `b` has the same value as the one pointed by `a`
`b = a;`	`b` also points to the object pointed by `a`

↦ A Pointer is a Variable Containing a Memory Address.

- Notation: *type* `*p;` -> "p is a pointer to *type*."
  Examples: `char *p`, `int *p`, `POINT *p`, `struct data *p`, etc.
- Accessing the data pointed to by p: `*p`;
- Accessing a field of a structure pointed to by p: `p->field` or `(*p).field`;
- Accessing the address of the variable `var`: `&var`;
- `void *p`: pointer to an undefined type (e.g., `malloc()`);
- `p = &data`; stores the address of data in p;
- `*p = data`; copies the value of the data variable into the variable pointed to by p;
- Pointer arithmetic (except for `void *`):
  - Adding/subtracting an integer: moves the pointer in memory by the size of the designated type;
    Examples:
    ```
    int *p; /* p is a pointer to integers */
    p = &i; /* p receives the address of i */
    p++; /* p points to the next integer in memory */
    ```
  - Subtracting pointers of the same type: returns the number of elements between the two addresses;
  - ! Any other operation is prohibited as it is meaningless!

↦ Dynamic allocation (heap) in C (#include <stdlib.h>)

- Allocation:
  - `void *malloc(size_t size);`
  - `void *calloc(size_t nmemb, size_t size);`
  - `void *realloc(void *ptr, size_t size);`
- De-allocation:
  - `void free(void *ptr)`

```
typedef struct point {
        double x, y;
} POINT;
POINT *p;
...
p = (POINT *)malloc(sizeof(POINT));
p->x = 0.0;
p->y = 0.0;
...
free(p);
```

↦ Array / Pointer Equivalence

- The notation [ ] can be used for both an array and an area designated by a pointer.

```
int* p;
p = (int*) malloc(sizeof(int)*100);
```

p[i] is equivalent to *(p+i) and p is equivalent to &p[0]

- Conversely, an array can be manipulated as a pointer (to the first element of the array).

```
int tab[100];
```

*(tab+i) is equivalent to tab[i] and tab is equivalent to &tab[0]

- Caution: sizeof(p) $\neq$ sizeof(tab). p has the size of a memory address: 4 bytes for a 32-bit processor, 8 bytes for a 64-bit processor. tab has the size of 100 integers: 400 bytes.

↦ Pointers on pointers

- It is often usefull to define pointers on pointers of a *type*.
- Example : Stock an image of 10×10 pixels coded in gray levels with integers.

```
int** p;
/* Allocation */
p = (int **) malloc(sizeof(int*) * 10);
for (i = 0; i < 10; i++) {
 *(p+i) = (int *) malloc(10 * sizeof(int));
}
...
/ *Désallocation */
for (i = 0; i < 10; i++) {
 free(*(p+i));
}
free(p);
```

TEST 4

Given the following definition:

```
int n = 4;
double p = 3.14;
```

How do we find the address of variables n, p?

TEST 4

Given the following definition:

```
int n = 4;
double p = 3.14;
```

How do we find the address of variables *n*, *p*?

```
int *pn = &n;
double *pp = &p;
```

How can we obtain 7.14 using the pointers *pn*, *pp*?

TEST 4

Given the following definition:

```
int n = 4;
double p = 3.14;
```

How do we find the address of variables *n*, *p*?

```
int *pn = &n;
double *pp = &p;
```

How can we obtain 7.14 using the pointers *pn*, *pp*?

```
*pp = *pp + *pn;
```

TEST 5

Given:

```
int a = 5, b = 7;
```

we want to write a function to swap the integers, such that by calling

```
swap(&a, &b);
```

we have $a = 7, \ b = 5$.
What is the prototype of the swap function?

TEST 5

Given:

```
int a = 5, b = 7;
```

we want to write a function to swap the integers, such that by calling

```
swap(&a, &b);
```

we have $a = 7,\ b = 5$.
What is the prototype of the swap function?

```
void swap(int * x, int * y);
```

How is swapping done?

TEST 5

Given:

```
int a = 5, b = 7;
```

we want to write a function to swap the integers, such that by calling

```
swap(&a, &b);
```

we have $a = 7,\ b = 5$.
What is the prototype of the swap function?

```
void swap(int * x, int * y);
```

How is swapping done?

```
void swap(int *x, int *y)
{
 int temp = *x;
 *x = *y;
 *y = temp;
}
```

TEST 6

Given:

```
int a [] = { 11, 24, 37 };
int *p;
p=a;
```

what is the result of the following operations?

```
(p+2) - p =
*p =
*(p+1) =
*(p+2) - *p =
```

Answer:

TEST 6

Given:

```
int a [] = { 11, 24, 37 };
int *p;
p=a;
```

what is the result of the following operations?

```
(p+2) - p =
*p =
*(p+1) =
*(p+2) - *p =
```

Answer:

```
(p+2) - p = 2
*p = 11
*(p+1) = 24
*(p+2) - *p = 26
```

TEST 7

Given the following strings of different length:

```
char str1[] = "hello";
char str2[] = "death star";
char str3[] = "obi wan";
```

declare an array of strings containing all of them:

TEST 7

Given the following strings of different length:

```
char str1[] = "hello";
char str2[] = "death star";
char str3[] = "obi wan";
```

declare an array of strings containing all of them:

```
char * strArray[] = { str1, str2, str3 };
```

Note: *strArray* contains only pointers, not the characters themselves!

TEST 8

```c
#include <stdio.h>
char* getMessage()
{
 char msg[] = "Pointers are fun";
 return msg;
}
int main ()
{
 char* string = getMessage();
 puts(string);
 return 0;
}
```

What is wrong with this code?

TEST 8

```c
#include <stdio.h>
char* getMessage()
{
 char msg[] = "Pointers are fun";
 return msg;
}
int main ()
{
 char* string = getMessage();
 puts(string);
 return 0;
}
```

What is wrong with this code?
The pointer is invalid after the variable passes out of scope!

TEST 9 (this is difficult)

```c
#include<stdio.h>
int main()
{
 static int i,j,k;
 int *(*ptr)[];
 int *array[3]={&i,&j,&k};
 ptr=&array;
 j=i+++k+10;
 ++(**ptr);
 printf("Output = %d",***ptr);
 return 0;
}
```

What is the output of this code?

TEST 9 (this is difficult)

```c
#include<stdio.h>
int main()
{
 static int i,j,k;
 int *(*ptr)[];
 int *array[3]={&i,&j,&k};
 ptr=&array;
 j=i+++k+10;
 ++(**ptr);
 printf("Output = %d",***ptr);
 return 0;
}
```

What is the output of this code?

```
Output = 10
```

TEST 9 (this is difficult)

```
#include<stdio.h>
int main()
{
 static int i,j,k;
 int *(*ptr)[];
 int *array[3]={&i,&j,&k};
 ptr=&array;
 j=i+++k+10;
 ++(**ptr);
 printf("Output = %d",***ptr);
 return 0;
}
```

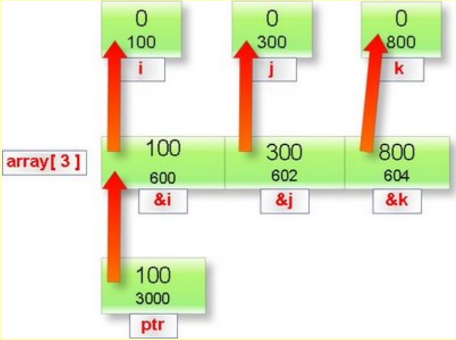
What is the output of this code?

```
Output = 10
```

Let's try to understand...

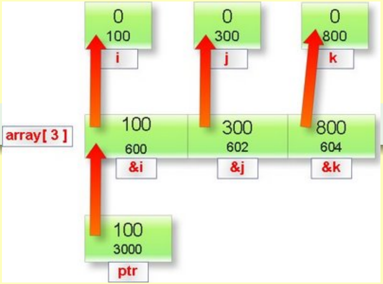
TEST 9 (this is difficult)

```
static int i,j,k;
int *(*ptr)[];
int *array[3]={&i,&j,&k};
ptr=&array;
```



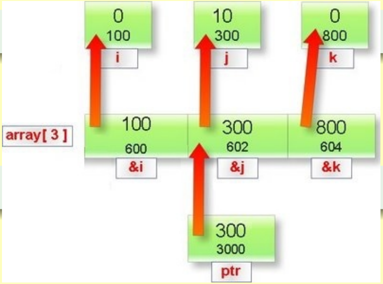
TEST 9 (this is difficult)

```
j=i+++k+10;
```



j=i++ +k +10
j=0 +0 +10 = 10
note that i++ is resolved after the sum!

```
++(**ptr);
```



uses the rules: array[0] = *(array+0)
++(**ptr) => **ptr = **ptr +1
=> ***ptr = *(array[1]) = *&j = j

```
printf("Output = %d",***ptr);
```

=> ***ptr = *(array[1]) = *&j = j = 10

Plan

Testing

- >
- <
- >=
- <=
- !=
- ==

Conditional Statements

```
if (condition) {
        // Code to execute if condition is true
} else {
        // Code to execute if condition is false
}
```

sectionThe `switch` Statement in C

The `switch` Statement

- The `switch` statement in C is used for multi-way branching.
- It provides an efficient way to handle multiple cases or choices in your code.
- The `switch` statement works with an expression and a set of cases to determine which code block to execute.

Basic switch Syntax

```
switch (expression) {
        case constant1:
        // Code to execute if expression equals consta
        break;
        case constant2:
        // Code to execute if expression equals consta
        break;
        // More cases can be added here
        default:
        // Code to execute if none of the cases match
}
```

- The `expression` is evaluated, and its value is compared to each case constant.
- If a match is found, the corresponding code block is executed.
- The `break` statement is used to exit the `switch` block after a case is executed.

Key Points about `switch`

- The `switch` statement is an efficient way to handle multiple cases based on a single expression.
- Each `case` constant is compared to the expression value, and the first matching case is executed.
- The `break` statement is used to exit the `switch` block after a case is executed to prevent fall-through.
- The `default` case is optional and is executed when none of the `case` constants match the expression.

```c
#include <stdio.h>

int main() {
        int choice = 2;

        switch (choice) {
                case 1:
                printf("You chose option 1.\n");
                break;
                case 2:
                printf("You chose option 2.\n");
                break;
                case 3:
                printf("You chose option 3.\n");
                break;
                default:
                printf("Invalid choice.\n");
        }

        return 0;
}
```

Loops

```
for (int i = 0; i < 5; i++) {
        // Code to repeat
}
```

```
while(condition){
 //Code to repeat until condition not true
}
```

```
do{
        //Code to repeat until condition not true
}while(condition)
```

Plan

Bitwise Operations

- Bitwise operations are used to manipulate individual bits of data.
- They are often used for low-level programming, such as working with hardware or optimizing algorithms.
- In C, there are several bitwise operators: `&`, `|`, `^`, `~`, «, and ».

```
result = a & b;
```

- The bitwise AND operator (&) sets each bit of the result to 1 if both corresponding bits of the operands are 1.
- Otherwise, it sets the bit to 0.

Bitwise OR (|)

```
result = a | b;
```

- The bitwise OR operator (|) sets each bit of the result to 1 if at least one of the corresponding bits of the operands is 1.
- It sets the bit to 0 if both corresponding bits are 0.

Bitwise XOR (ˆ)

```
result = a ^ b;
```

- The bitwise XOR operator (ˆ) sets each bit of the result to 1 if the corresponding bits of the operands are different (one is 1, and the other is 0).
- It sets the bit to 0 if both corresponding bits are the same.

Bitwise NOT (˜)

```
result = ~a;
```

- The bitwise NOT operator (˜) inverts each bit of the operand, changing 1s to 0s and 0s to 1s.

Bitwise Left Shift (<<) and Right Shift (>>)

```
result = a << n; // Left shift
result = a >> n; // Right shift
```

- The left shift operator (<<) shifts the bits of the operand to the left by `n` positions, filling with zeros on the right.
- The right shift operator (>>) shifts the bits to the right by `n` positions, filling with zeros or the sign bit on the left, depending on the data type.

Bitwise Operations Summary

- Bitwise operations manipulate individual bits of data.
- They are used for tasks like setting, clearing, or toggling specific bits.
- Understanding bitwise operations is essential for low-level programming and bit-level optimizations.

Plan

What is Preprocessing?

- Preprocessing is an initial step in the compilation of C programs.
- It involves processing directives that begin with a hash symbol (#) before the actual compilation.
- The C preprocessor performs text replacement and file inclusion.

- Preprocessor directives start with # symbol and are executed before compilation.

```
#include <stdio.h>
#define MAX_VALUE 100

int main() {
        int num = MAX_VALUE;
        printf("The maximum value is %d\n", num);
        return 0;
}
```

- `#include` - Includes a header file in the source code.
- `#define` - Defines macros and constants.
- `#ifdef`, `#ifndef`, `#if`, `#else`, `#elif`, `#endif` - Conditional compilation.
- `#error` - Generates an error message during preprocessing.
- `#pragma` - Provides compiler-specific instructions.

Header File Inclusion

- The `#include` directive is used to include header files.
- Header files contain declarations and definitions that are needed in your program.

```
#include <stdio.h>
#include "myheader.h"
```

Macros and Constants

- The `#define` directive is used to create macros and constants.

```
#define MAX_VALUE 100
#define SQUARE(x) ((x) * (x))

int main() {
        int num = MAX_VALUE;
        int square = SQUARE(5);
        return 0;
}
```

Preprocessing Summary

- Preprocessing is the initial step in compiling C programs.
- Preprocessor directives, starting with #, perform tasks like file inclusion, macro definition, and conditional compilation.
- Proper use of preprocessing can make your code more modular and maintainable.

Plan

I/O in C

- Input and Output (I/O) operations are essential for interacting with users and external data in C programs.
- C provides a set of standard I/O functions for reading and writing data.

Standard I/O Functions

- `printf()` - Used for formatted output to the console.
- `scanf()` - Used for formatted input from the console.
- `getchar()` and `putchar()` - Used for character-based input and output.
- `fgets()` and `fputs()` - Used for reading and writing strings.
- `fread()` and `fwrite()` - Used for binary file I/O.

Formatted Output: printf()

```c
#include <stdio.h>

int main() {
        int num = 42;
        printf("The answer is %d\n", num);
        return 0;
}
```

- printf() allows you to format and display data on the console.
- Format specifiers (e.g., %d, %s) are used to specify the type and format of the data to be printed.

Formatted Input: scanf()

```c
#include <stdio.h>

int main() {
        int num;
        printf("Enter an integer: ");
        scanf("%d", &num);
        printf("You entered: %d\n", num);
        return 0;
}
```

- scanf() is used to read and parse formatted input from the console.
- Format specifiers are used to specify the expected format of the input data.

File I/O: FILE and fopen()

```c
#include <stdio.h>

int main() {
        FILE *file = fopen("example.txt", "w");
        if (file != NULL) {
                fprintf(file, "Hello, File!\n");
                fclose(file);
        }
        return 0;
}
```

- To work with files, you need to declare a `FILE` pointer and use `fopen()` to open a file for reading or writing.

File I/O: Reading and Writing

```
#include <stdio.h>

int main() {
        FILE *file = fopen("example.txt", "r");
        if (file != NULL) {
                char buffer[100];
                while (fgets(buffer, sizeof(buffer), f
                        printf("%s", buffer);
                }
                fclose(file);
        }
        return 0;
}
```

- Use `fgets()` for reading lines from a file, and `fprintf()` for writing to a file.

I/O Management Summary

- Input and Output (I/O) operations in C are crucial for interacting with users and external data.
- Standard I/O functions like `printf()` and `scanf()` enable formatted input and output.
- File I/O involves opening files using `fopen()`, reading with `fgets()`, and writing with `fprintf()`.
- Proper error handling is essential when working with files and I/O functions.

Plan

Functions in C

- Functions are a fundamental part of C programming.
- They allow you to break down your code into smaller, reusable pieces.
- Functions can take parameters and return values.

Defining a Function

```
// Function declaration
int add(int a, int b);

int main() {
        int result = add(5, 3);
        return 0;
}

// Function definition
int add(int a, int b) {
        return a + b;
}
```

- A function is declared with a return type, name, and parameter list.
- The function is defined elsewhere in the code with the same name, return type, and parameter list.

Passing Pointers to Functions

```c
#include <stdio.h>

void swap(int *x, int *y) {
        int temp = *x;
        *x = *y;
        *y = temp;
}

int main() {
        int a = 5, b = 10;
        swap(&a, &b);
        printf("a: %d, b: %d\n", a, b);
        return 0;
}
```

- Functions can accept pointers as arguments to modify the values of variables in the calling function.
- In this example, `swap()` swaps the values of two integers using pointers.

Pointer Passing in Functions

- Passing pointers to functions allows functions to modify variables in the calling function.
- It is a way to achieve pass-by-reference behavior in C.
- Be cautious with pointer passing to prevent unintended side effects.

Example: Using Pointers in Functions

```c
#include <stdio.h>

void modifyArray(int *arr, int size) {
        for (int i = 0; i < size; i++) {
                arr[i] *= 2;
        }
}

int main() {
        int numbers[] = {1, 2, 3, 4, 5};
        int size = sizeof(numbers) / sizeof(numbers[0]
        modifyArray(numbers, size);
        for (int i = 0; i < size; i++) {
                printf("%d ", numbers[i]);
        }
        return 0;
}
```

- In this example, the `modifyArray()` function modifies an array using pointers.

Functions and Pointer Passing Summary

- Functions in C allow you to encapsulate code for reuse and readability.
- Pointers can be passed to functions to modify variables in the calling function, achieving pass-by-reference behavior.
- Proper handling of pointers in functions is essential to avoid unintended side effects and memory issues.

Plan

↦ **Creating an Executable Program from C Code Involves Several Rules.**

- One and only one of the used source files contains a `main()` function.
- This function is the program's main function.
- All processing starts from `main()`, either directly or by calling other functions.
- These other functions can be:
  - Standard and generally portable C functions.
  - Functions written by the programmer.

Three Steps

1. **Preprocessing**
2. **Compilation**
3. **Linking**

↦ **The Compilation and Linking Stages Are Preceded by a Preprocessing Step. The Preprocessor Allows Pre-Compilation and Linking Manipulations.**

- Including files using the `#include` directive:
  - This directive allows the inclusion of header files (e.g., `.h` files).
  - These files allow referencing functions whose symbols will be resolved during linking.
  - They also allow the definition of types, structures, and more.
- Replacing code snippets with constants or macros using the `#define` directive;

```
#define PI 3.14159
#define norm(x,y) (sqrt(pow(x,2.0)+pow(y,2.0)))
#define myfabs(x) ((x < 0) ?  -x :  x)
```

- Conditionally including code snippets during compilation using `#ifdef...#endif` or `#ifndef...#endif`;

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
....conditionally included block...
#endif
```

↦ The Program Production Involves a Compilation Step.

- It allows the generation of an object file `.o` from a source file `.c`.
- The object file contains a "translation" of the C source code into code understandable by the processor (assembly code).

- GCC (GNU C Compiler) created in 1985 by Richard Stallman.
- `gcc -c foo.c` produces the object file `foo.o`.
- Commonly used options with `gcc` during the compilation stage include:
  - `-g` to enable debugging.
  - `-Wall` to display all warnings.
  - `-O0` to enforce code optimization off (default).
  - ...

↦ Compilation Must Be Followed by a Linking Step.

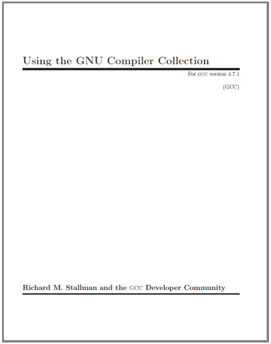
- It allows the gathering of all object files that compose the program.
- Its role is to resolve all references to undefined symbols in different object files.
- The GNU linker is the program `ld`, which is usually called implicitly through `gcc`.
- `gcc -o my_exec main.o titi.o tutu.o tata.o` produces the executable `my_exec`.
- Object files can be grouped into static libraries `.a` using the `ar` and `ranlib` commands:

```
ar cr libtoto.a titi.o tutu.o tata.o
ranlib libtoto.a
```

- Linking to a library is done as follows: `gcc -o my_exec main.o -L ./lib -ltoto`.
- Linking to the standard math library is done with `-lm`, where `libm.a` is the name of the library.
- The option `-L library_path` can be used to specify library paths that are not standard C libraries (`/usr/lib`).
- There are also dynamically linked libraries (`.so` or `.dll` on Windows).

Common options:

- Generic
  - `-S` stop after compilation, do not assemble
  - `-E` stop after preprocessing, do not compile
  - `-o` *file* place output in file *file* (default *a.out*)
- Optimization
  - `-O0` no optimization
  - `-O1` optimize and minimize compile time
  - `-O2` more costly optimization
  - `-Os` optimize for code size
- C language
  - `-ansi` , `-std=c90` , `-std=iso9899:1990` to select the standard C
  - `-pedantic` for all the diagnostics required by the specific standard
  - `-pedantic-errors` to treat warning as errors
- Debugging and errors
  - `-g` enables generation of debugging information, that can be used by `gdb` (works with `-O0`)
  - `-Werr` make all warnings into errors
  - `-Wall` enables all (main) warnings
  - `-Wextra` enables extra warnings not enabled by `-Wall` (for example `-Wsign-compare`)
- Threads
  - `-pthreads` support multithreading using POSIX threads library
- Linking
  - `-l`*library* search the library named *library* when linking (default lib*library*.a), for example `-lm` links the math library
  - `-L`*dir* add *dir* to the list of directories to be searched for `-l`



Using the GNU Compiler Collection

For GCC version 4.7.1

(GCC)

Richard M. Stallman and the GCC Developer Community

the manual can be downloaded from gcc.gnu.org

Example: compilation

bidule.c

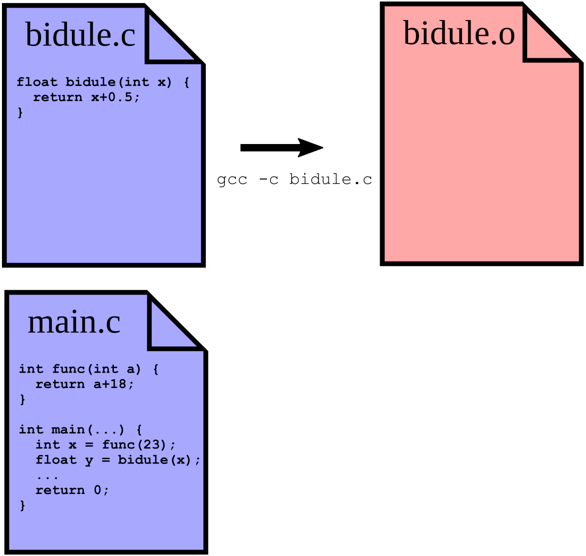
```
float bidule(int x) {
  return x+0.5;
}
```

main.c

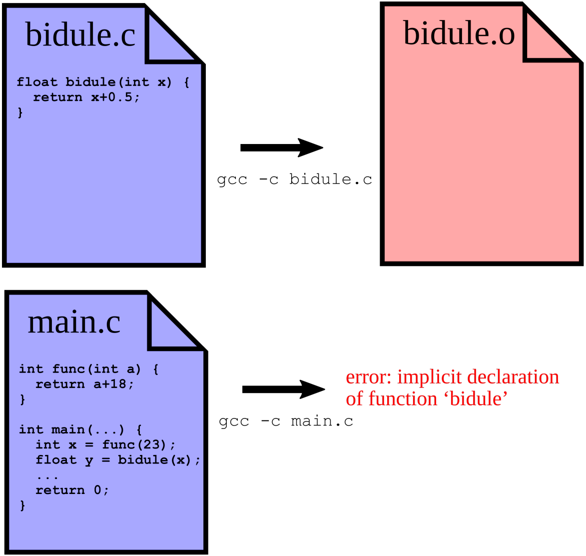
```
int func(int a) {
  return a+18;
}

int main(...) {
  int x = func(23);
  float y = bidule(x);
  ...
  return 0;
}
```

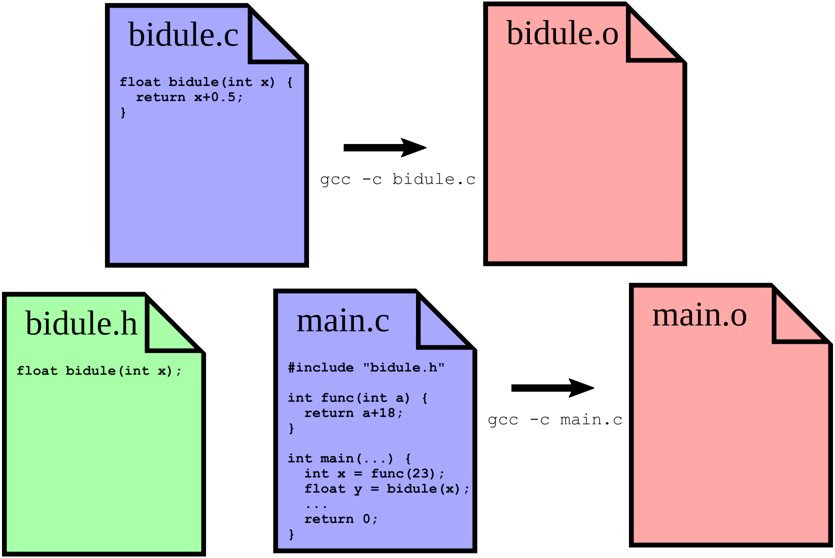
Example: compilation



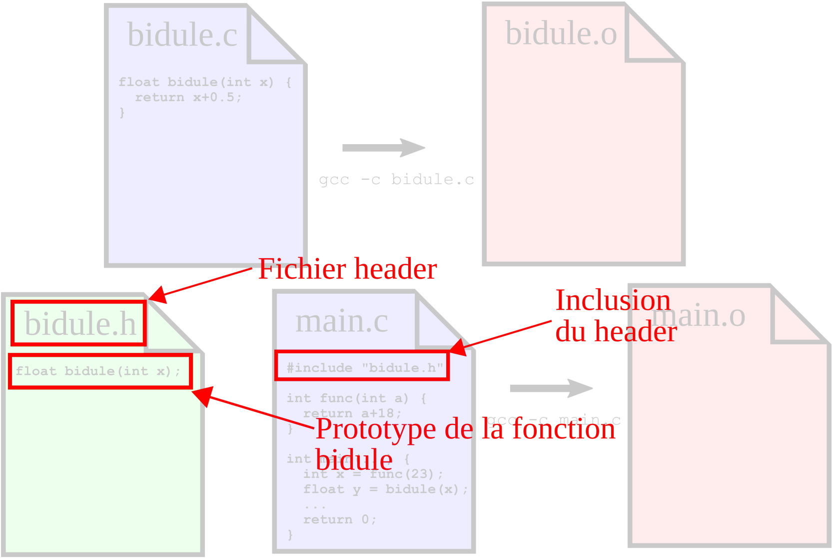
Example: compilation



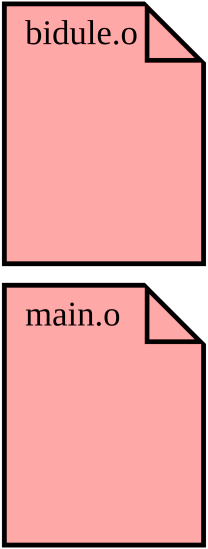
Example: compilation



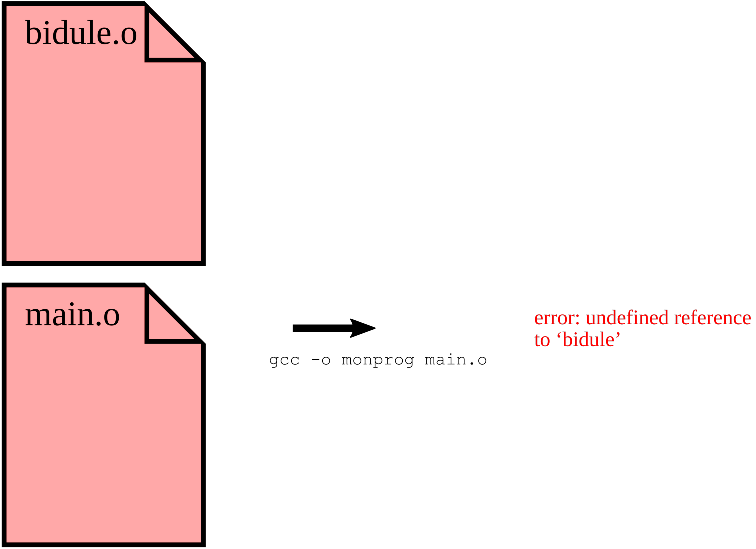
Example: compilation



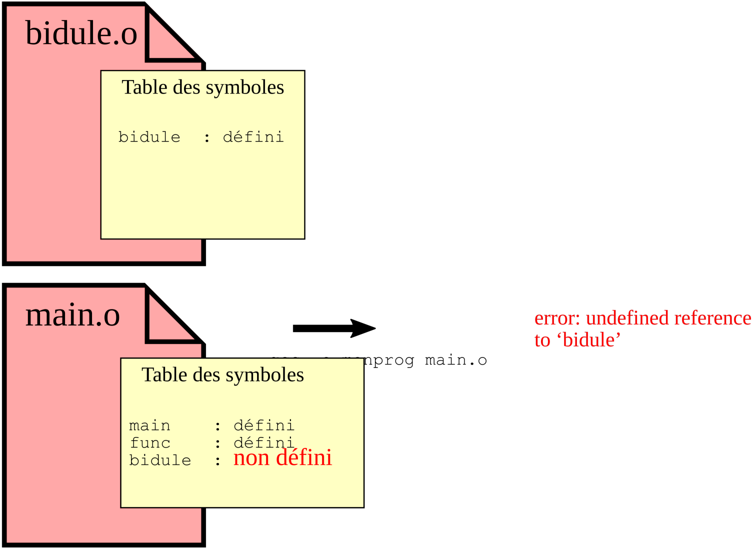
Example: linking



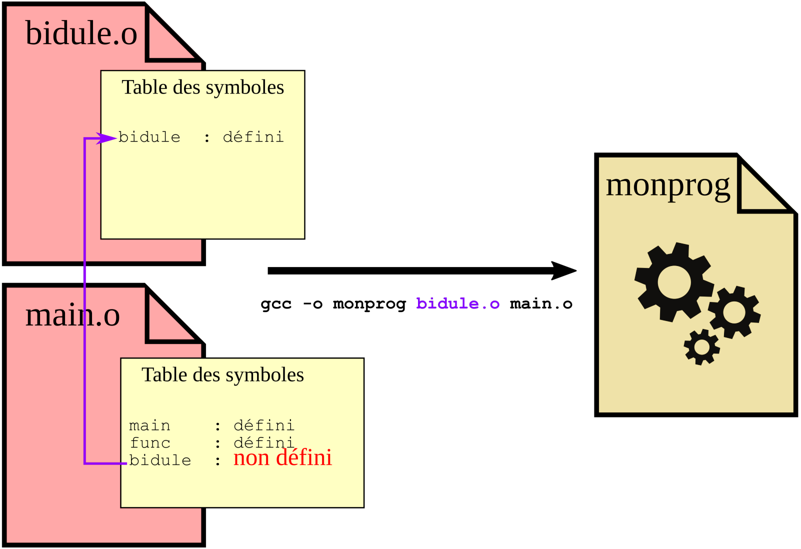
Example: linking



Example: linking



Example: linking



TEST 3

Editing: test.c

```
/* a test */
#include<stdio.h>
int main(int argc, char* argv[])
{
 puts("Hello students!"); //output text to stdout and end line
 return 0;
}
```

How do we compile?

TEST 3

Editing: test.c

```
/* a test */
#include<stdio.h>
int main(int argc, char* argv[])
{
 puts("Hello students!"); //output text to stdout and end line
 return 0;
}
```

How do we compile?

```
gcc -Wall test.c -o test
```

Or:

```
gcc -Wall -c test.c
gcc -Wall -o test test.o
```

↦ Exemple simple

- Création de l'exécutable `my_exec`
- Fonction `main()` déclarée dans `main.c`
- `main()` appelle les fonctions `titi()`, `tata()` et `tutu()` respectivement déclarées dans `titi.c`, `tata.c` et `tutu.c`
- `titi()` et `tata()` appellent `tutu()`

↪ Comment générer l'exécutable `my_exec` ?
↪ Avec ou sans bibliothèque ?

↦ Exemple simple

- Création de l'exécutable `my_exec`
- Fonction `main()` déclarée dans `main.c`
- `main()` appelle les fonctions `titi()`, `tata()` et `tutu()` respectivement déclarées dans `titi.c`, `tata.c` et `tutu.c`
- `titi()` et `tata()` appellent `tutu()`

↪ Comment générer l'exécutable `my_exec` ?
↪ Avec ou sans bibliothèque ?

Solution basique

Compilation

» gcc -g -c titi.c
» gcc -g -c tutu.c
» gcc -g -c main.c
» gcc -g -c tata.c

Edition de liens

» gcc -g -o my_exec main.o tutu.o tata.o titi.o

↦ Exemple simple

Solution basique

Compilation

» gcc -g -c titi.c
» gcc -g -c tutu.c
» gcc -g -c main.c
» gcc -g -c tata.c

Edition de liens

» gcc -g -o my_exec main.o tutu.o tata.o titi.o

Solution utilisant une bibliothèque

Compilation

» gcc -g -c titi.c
» gcc -g -c tutu.c
» gcc -g -c main.c
» gcc -g -c tata.c

Création de la bibliothèque

» ar cr libtoto.a tutu.o titi.o tata.o
» ranlib libtoto.a

Edition de liens

» gcc -g -o my_exec2 main.o -L . -ltoto

Simple Example... but Still...

Compilation

```
» gcc -g -c titi.c
» gcc -g -c tutu.c
» gcc -g -c main.c
» gcc -g -c tata.c
```

Library Creation

```
» ar cr libtoto.a tutu.o titi.o tata.o
» ranlib libtoto.a
```

Linking

```
» gcc -g -o my_exec2 main.o -L . -ltoto
```

Remarks

- Long and tedious for more than 2 files to compile (projects often consist of several hundred files)!
- When a source file is modified, it's not necessary to recompile everything. For efficiency, you need to know the dependencies and only recompile and link what is necessary. This is not feasible beyond 5 source files from a human perspective.

↪ **There are tools available to automate the program creation process: make, and for larger projects, CMake, autotools, ...**

Plan

↦ `make` **is a program that automates all the necessary operations to obtain an executable**

- The instructions executed by `make` are placed in a `Makefile` file.
- Using file access dates, `make` can determine which files need to be recompiled or not → **dependency management**.
- The only strictly necessary knowledge for writing "simple" `Makefile` files is that of writing rules:
  - TARGET: DEPENDENCIES
    COMMAND
    ...
  - `TARGET` can represent the name of a file to produce (a `.o` file, an executable...) or more generally, any name (`question2`, for example).
  - `DEPENDENCIES` represents the set of rules that must have been executed and/or the set of files that must exist in order to run the `COMMAND`.
- `make` is actually much more general than this. For example, this document was created using `make`. (general framework of C compilation).

↪ **A good reference document on `make` can be found at this address: http://www.laas.fr/~matthieu/cours/make**

↪ **Make automates the compilation chain... CMake, Autotools, and some IDEs automate the generation of Makefiles (or any other standardized mechanism for efficient description of the compilation chain)**

Plan

What is CMake?

- CMake is an open-source, cross-platform build system and project management tool.
- It simplifies the process of building and managing C projects.

Why Use CMake?

- Advantages of using CMake for C projects:
  - Cross-platform compatibility.
  - Simplifies the build process.
  - ...

Basic CMake Concepts

- CMake uses CMakeLists.txt files to define project configurations.
- Key concepts include CMakeLists.txt, targets, source files, and build directories.

CMake Example Directory Structure

- Example directory structure:
  - src/ (source code)
  - cmake/ (CMake scripts)
  - build/ (build files)

CMakeLists.txt File

- Contents of a CMakeLists.txt file:
  - Set project details.
  - Specify source files.
  - Create targets.

- Example of a simple C program.
- Corresponding CMakeLists.txt file for building the program.

Running CMake to Generate Build Files

- How to run CMake to generate platform-specific build files (e.g., Makefiles, Visual Studio project files).
- Use of the CMake command-line tool.

Building the Project

- Using the generated build files to build the C project.
- Compiling the code with CMake.

Cross-Platform Building

- CMake's ability to generate build files for different platforms (Windows, Linux, macOS).
- Importance of maintaining platform independence.

Example for C Project

```
# Minimum required CMake version
cmake_minimum_required(VERSION 3.0)

# Project name and description
project(MyCProject C)

# Specify the source files for the project
set(SOURCES
src/main.c
src/utils.c
)

# Specify the header files for the project
set(HEADERS
include/utils.h
)

# Create an executable target
add_executable(my_c_program ${SOURCES} ${HEADERS})

# Specify include directories
target_include_directories(my_c_program PRIVATE include)

# Additional compiler options (optional)
# target_compile_options(my_c_program PRIVATE -Wall -Wextra)
```

- Key points about CMake and its usefulness for C projects.
- Encourage developers to explore CMake for managing their projects efficiently.

↦ Simple Example

- The following example produces an executable `myprog` from the files `main.o` and `file1.o`.
- These two `.o` files depend on their respective source files. Additionally, `main.o` depends on `file1.h`.
- The `clean` rule deletes `.o` files. The `vclean` rule applies the `clean` rule and then deletes the executable `myprog`.

```
myprog:  main.o file1.o
	gcc -o myprog main.o file1.o

file1.o:  file1.c
	gcc -c file1.c

main.o:  main.c file1.h
	gcc -c main.c

clean:
	rm -f *.o

vclean:  clean
	rm -f myprog
```

Plan

↦ Debugging Tools

- Syntax or conceptual errors can cause the compilation and/or linking phase to fail. Furthermore, once the executable is generated, its behavior may not be as expected: incorrect results, non-conforming behavior, segmentation faults, etc.
- This requires the use of debugging tools:
  - The simplest one: the function `int printf(const char *format, ...)` which allows displaying strings on the screen.
  - The function `void assert(scalar expression)` which terminates the program in case of failure (0) of the test.
  - The debugger `gdb` (and optionally its graphical interface `ddd`) which allows step-by-step program execution while visualizing the evolution of variable values.

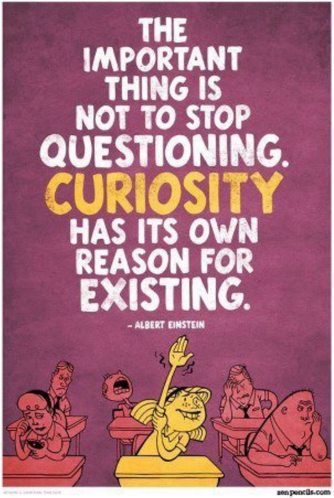
```
» gdb ./my_exec
(gdb) run main_arguments
...
```

  - The tool `valgrind` which, among other things, checks whether dynamically allocated memory is properly freed by the program (there are many other possibilities offered by `valgrind`).

```
» valgrind ./my_exec run main_arguments ...
```

  - The `gdb` and `valgrind` tools require compilation and linking with the -g option.

Questions?


THE IMPORTANT THING IS NOT TO STOP QUESTIONING. CURIOSITY HAS ITS OWN REASON FOR EXISTING.
- ALBERT EINSTEIN
zenpencils.com