C Macros, `const`, `volatile` and `inline`

Lecture 7

Christian A. Pagot

Universidade Federal da Paraíba
Centro de Informática

C Preprocessor

*The **C preprocessor** is a **macro** processor that is used **automatically** by the C compiler to transform your program **before** compilation.*

GNU documentation

Macro Examples

```
#ifdef _DEBUG
    #define GL_CHECK(stmt) do { \
            stmt; \
            CheckOpenGLError(#stmt, __FILE__, __LINE__); \
        } while (0)
#else
    #define GL_CHECK(stmt) stmt
#endif
```

Stackoveflow.com

cl.h

```
/* Error Codes */
#define CL_SUCCESS                                  0
#define CL_DEVICE_NOT_FOUND                         -1
#define CL_DEVICE_NOT_AVAILABLE                     -2
#define CL_COMPILER_NOT_AVAILABLE                   -3
#define CL_MEM_OBJECT_ALLOCATION_FAILURE            -4
...
```

OpenCL

```
#define ARRAY_SIZE(a) (sizeof(a) / sizeof(*a))
```

Stackoveflow.com

C Preprocessor Directives

The list below includes some of the directives supported by the C preprocessor:

- `#define`
- `#undef`
- `#include`
- `#ifdef ... #else ... #endif`
- `#ifndef ... #else ... #endif`
- among others.

#include

- **Inserts** the contents of an **external** file into **current source** code at the point where **#include** is invoked.
- Example

maininclude.c

```
#include "stdio.h"

int main( void )
{
    printf("Hellow world!\n");

    return 0;
}
```

Includes the contents of `stdio.h` header file into the `maininclude.c` source code.

Take a look at the effects of the `#include` above with:

```
~$ gcc -std=c99 -E maininclude.c > maininclude.i
```

#define

- Create macros.
  - **#define** is followed by the **name** of the **macro** and the token sequence it should be an abbreviation for.
- Example

Macro defined with #define preprocessing directive

maindefine.c

```
#define VALUE 99

int main( void ) {
    int x;

    x = VALUE;

    return x;
}
```

Macro defined as a GCC command line argument

```
~$ gcc -D VALUE=99 ...
```

maindefine.c

```
int main( void ) {
    int x;

    x = VALUE;

    return x;
}
```

#ifdef ... #else ... #endif

Allows the compiler to conditionally remove lines of code based on the existence (or not) of a macro.

Macro defined with `#define` preprocessing directive

mainifdef.c

```
#define VALUE 99

int main( void ) {
    int x;

#ifdef VALUE
    x = VALUE;
#else
    x = 255;
#endif

    return x;
}
```

Macro defined as a GCC command line argument

```
~$ gcc -D VALUE=99 ...
```

mainifdef.c

```
int main( void ) {
    int x;

#ifdef VALUE
    x = VALUE;
#else
    x = 255;
#endif

    return x;
}
```

Include Guards

Type definition.

Suppose the following C program:

maintriangle.c

```
struct Vertex {
    float position[3];
    float normal[3];
};

struct Triangle {
    struct Vertex vertices[3];
};

int main( void ) {
    struct Vertex v;

    struct Triangle t;

    return v.position[0] + t.vertices[0].position[0];
}
```

vertex.h

```
struct Vertex {
    float position[3];
    float normal[3];
};
```

triangle.h

```
#include "vertex.h"

struct Triangle {
    struct Vertex vertices[3];
};
```

main.c

```
#include "vertex.h"
#include "triangle.h"

int main( void ) {
    struct Vertex v;

    struct Triangle t;

    return ...
}
```

Include Guards

vertex.h

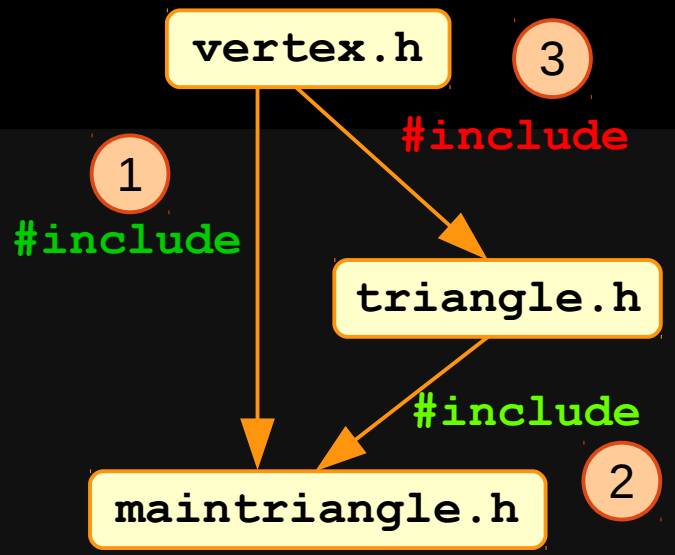
```
struct Vertex {
    float position[3];
    float normal[3];
};
```

triangle.h

```
#include "vertex.h"

struct Triangle {
    struct Vertex vertices[3];
```

maintriangle.c

```
#include "vertex.h"
#include "triangle.h"

int main( void ) {
    struct Vertex v;

    struct Triangle t;

    return ...
}
```



Let's build `maintriangle`:

```
~$ gcc -Wall -Werror -std=c99 -E main.c > main.i
~$ gcc -Wall -Werror -std=c99 -S main.i
In file included from triangle.h:1:0,
                 from main.c:2:
vertex.h:4:8: error: redefinition of ‘struct Vertex’
 struct Vertex {
        ^
In file included from main.c:1:0:
vertex.h:4:8: note: originally defined here
 struct Vertex {
        ^
```

How to solve that?

Include Guards

vertex.h (old)

```
struct Vertex {
    float position[3];
    float normal[3];
};
```

vertex.h (new)

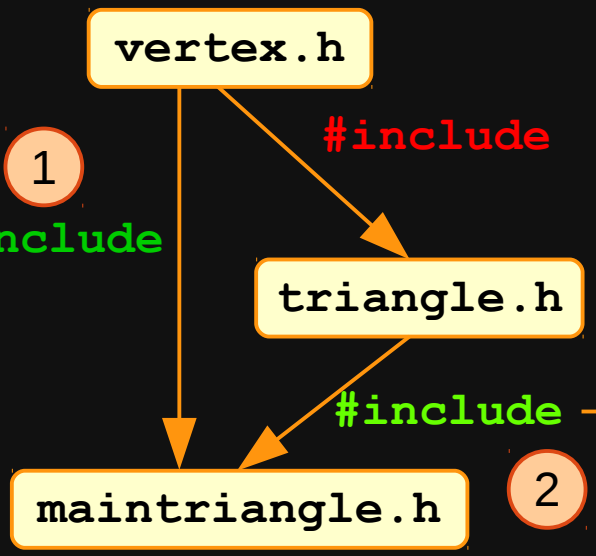
```
#ifndef VERTEX_H_
#define VERTEX_H_

struct Vertex {
    float position[3];
    float normal[3];
};

#endif // VERTEX_H_
```

Include guards!

Guarded code Block (it won't be included twice)!

Every header should be guarded with inc. guards (even `triangle.h`)!

`maintriangle.c` will `#include vertex.h`. As `vertex.h` is read into `maintriangle.c`, its preprocessing directives are executed and (since it was not previously defined) the macro `VERTEX_H_` is first defined.



As soon as `triangle.h` starts to be read into `maintriangle.c` , it `#include vertex.h`. The preprocessor will thus start reading `vertex.h` into `triangle.h`. As soon as it finds the preprocessor directive `#ifndef`, it ignores the contents of the conditional block until `#endif`.

Function-like Macros

It is a macro whose name is followed by parentheses.

- These macros can take arguments!

mainfmacro.c

```
#define OFFSET_VALUE 10

#define OFFSET_FUNCTION( val ) val + OFFSET_VALUE

int main( int argc, char *argv[] )
{
    int x;

    if ( argc > 1)
        x = 2;
    else
        x = 1;

    int y = OFFSET_FUNCTION( x );

    return y;
}
```

Creates a function-like macro called `OFFSET_FUNCTION` which takes `val` as argument.

Macro Pitfalls

mainfmacro.c

```
#define OFFSET_VALUE 10

#define OFFSET_FUNCTION( val ) val + OFFSET_VALUE

int main( int argc, char *argv[] )
{
    int x;

    if ( argc > 1)
        x = 2;
    else
        x = 1;

    int y = 3 * OFFSET_FUNCTION( x );

    return y;
}
```

3 * the result of OFFSET_FUNCTION

mainfmacro.i

```
# 1 "mainmacros.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "mainmacros.c"

int main( int argc, char *argv[] )
{
    int x;

    if ( argc > 1)
        x = 2;
    else
        x = 1;

    int y = 3 * x + 10

    return y;
}
```

According to the code above, the value of **OFFSET_FUNCTION** is multiplied by 3. However, the **actual result** is **different. Why?**

```
~$ gcc -Wall -Werror -pedantic -std=c99 -O0 -E mainmacros.c > mainmacros.i
```

A Solution for this Problem Instance

mainfmacro.c (old)

```
#define OFFSET_VALUE 10

#define OFFSET_FUNCTION( val ) val + OFFSET_VALUE

int main( int argc, char *argv[] )
{
    ...
```

mainfmacro.c (new)

```
#define OFFSET_VALUE 10

#define OFFSET_FUNCTION( val ) ( val + OFFSET_VALUE )

int main( int argc, char *argv[] )
{
    ...
```

```
~$ gcc -Wall -Werror -pedantic -std=c99 -O0 -E mainfmacro.c > mainfmacro.i
```

mainfmacro.i

```
...
int main( int argc, char *argv[] )
{
    int x;

    ...

    int y = 3 * ( x + 10 );

    return y;
}
```

Now the expression will be correctly evaluated!

`const` type Qualifier

`const` indicates to the compiler that the **content** of the qualified object **will not** be changed.

- The value of a `const` variable can be eventually changed, but not by the current program.
  - Sometimes it is said that `const` variables would be more accurately described as read-only variables (more on that later)!

`const` Variable

mainconst1.c

```
int i = 10;

int main( void )
{
    return i;
}
```

If the value of `i` is not going to be modified along the program, we can declare it as `const`.

mainconst2.c

```
const int ci = 20;

int main( void )
{
    ci = 30;


    int i = ci;

    return i + ci;
}
```

`const` variables must be initialized!

Any attempt to modify the value of `ci` will make the compiler raise an error.

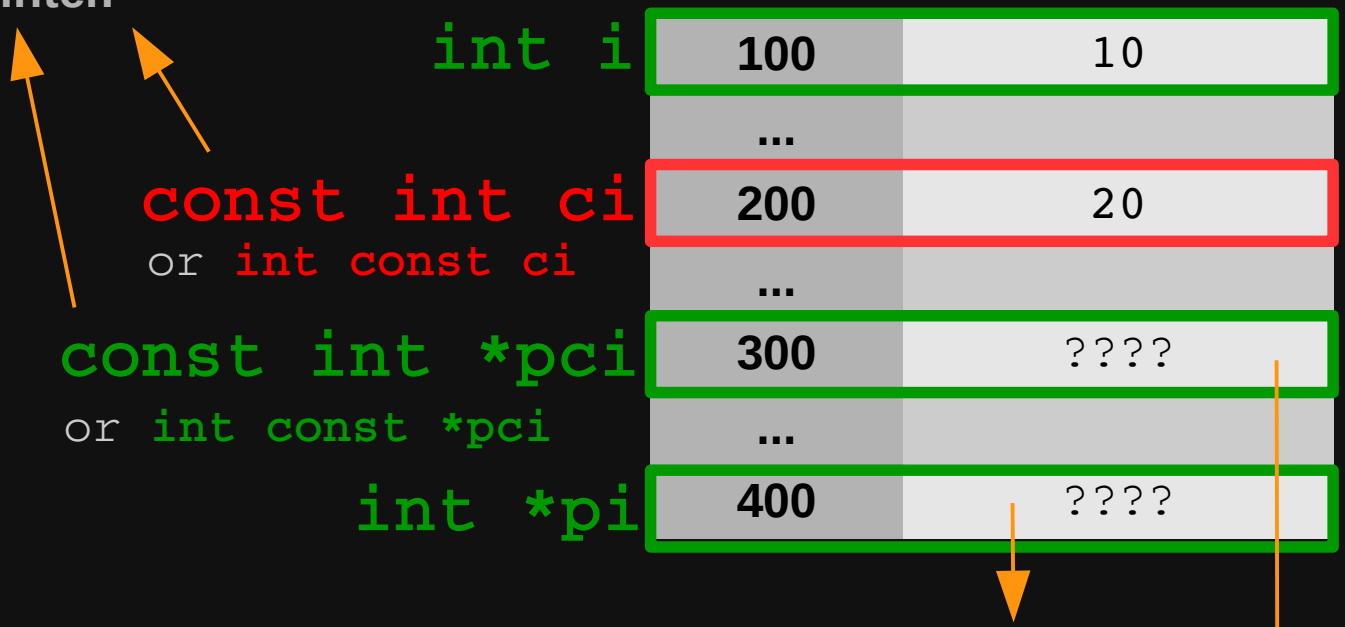
In an attribution, the `const` value can be freely copied into other non-const variables.

Pointer to `const`

mainconst3.c

```
int i = 10;

const int ci = 20;

const int *pci;

int *pi;

int main( void )
{
    pci = &i;

    pci = &ci;

    pci = pi;

    pi = pci;

    return ...
}
```

The value of the pointed variable cannot be modified through the pointer.



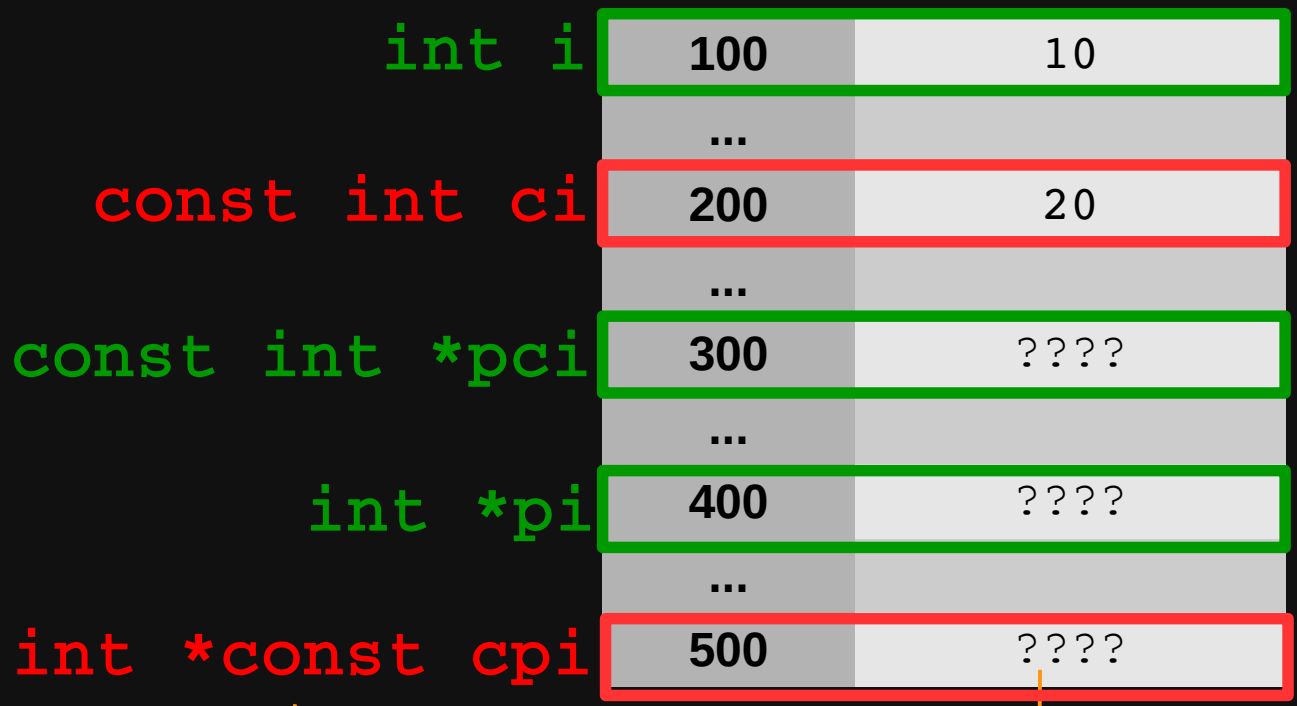
May contain only the address of a non-const variable.

Might contain the address of a `const` or non-const variable.

`const` Pointer

mainconst4.c

```
int i = 10;

const int ci = 20;

const int *pci;

int *pi;

int main( void )
{
    int *const cpi;

    int *const cpi = &i;

    int *const cpi = &ci;

    int *const cpi = pi;

    int *const cpi = pci;

    return ...
}
```



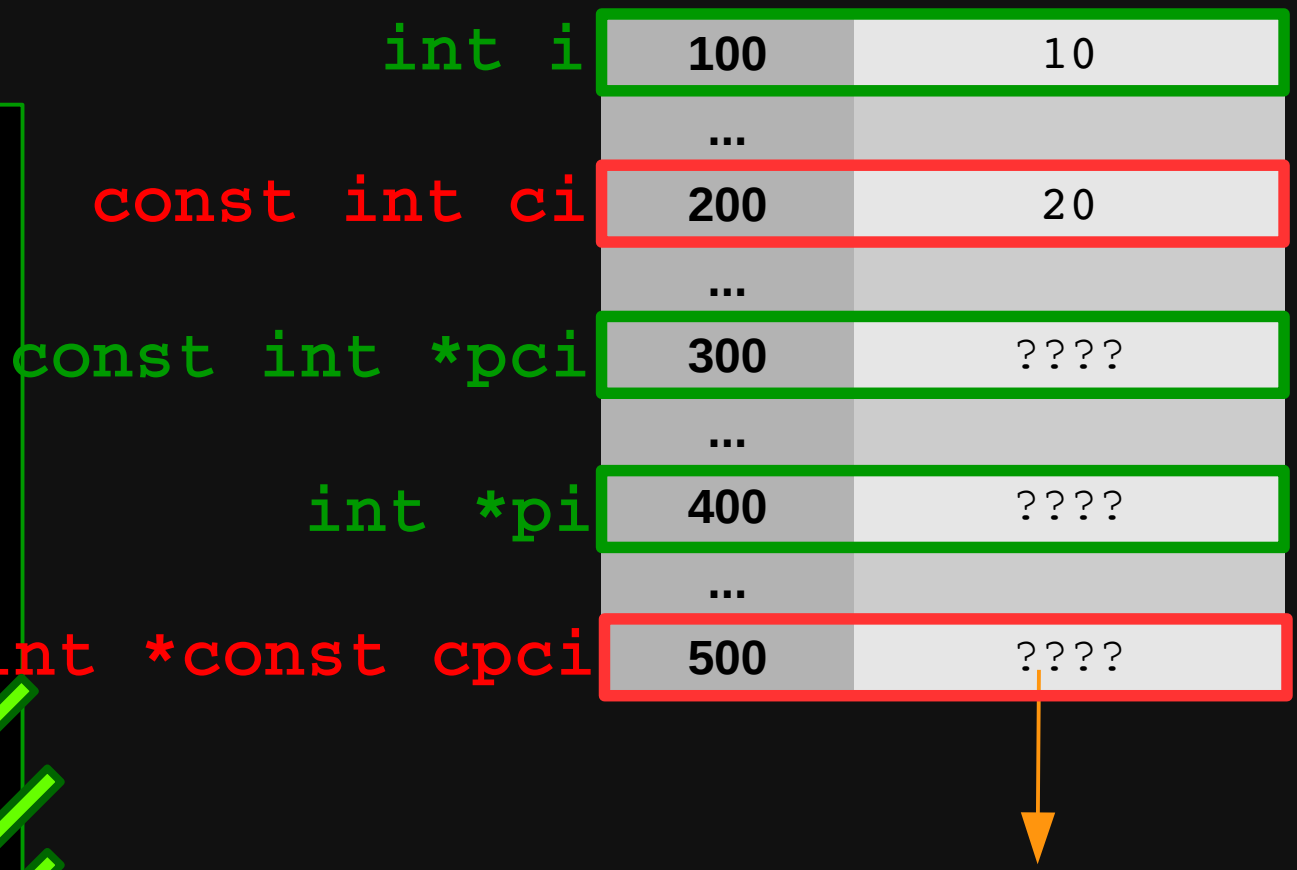
The value of the pointer cannot be modified to point to something else.

May contain only the address of a non-const variable.

`const` Pointer to `const`

mainconst5.c

```
int i = 10;

const int ci = 20;

const int *pci;

int *pi;

int main( void )
{
 const int *const cpci;

 const int *const cpci = &i;

 const int *const cpci = &ci;

 const int *const cpci = pci;

 const int *const cpci = pi;

 return ...
}
```



May contain the address of `const` and non-const variable.

`const` vs. Function Parameters

mainfconst1.c

```
int f_i( int a )
{
    return a;
}
...
```

a is a `int` parameter passed by value.

mainfconst1.c

```
...
int i = 10;
const int ci = 20;
int *pi;
const int *pci;

int main( void )
{
    int *const cpi = &i;
    const int *const cpci = &i;

    i = f_i( i );

    i = f_i( ci );

    i = f_i( *pi );

    i = f_i( *cpi );

    i = f_i( *pci );

    i = f_i( *cpci );
    ...
}
```

`f()` receives copies of the integer arguments.

`f()` receives copies of the integer values that are being pointed. Thus, it does not matter if the pointed value is `const` because the function can only alter its copy!

`const` vs. Function Parameters

mainfconst2.c

```
int f_ci( const int a )
{
    return a;
}
...
```

a is a `const int` parameter passed by value.

`f()` receives copies of the integer arguments. The copy will be treated as `const` within `f()`.

`f()` receives copies of the integer values that are being pointed. The parameter will be treated as `const` within the function.

mainfconst2.c

```
...
int i = 10;
const int ci = 20;
int *pi;
const int *pci;

int main( void )
{
    int *const cpi = &i;
    const int *const cpci = &i;

    i = f_ci( i );

    i = f_ci( ci );

    i = f_ci( *pi );

    i = f_ci( *cpi );

    i = f_ci( *pci );

    i = f_ci( *cpci );
    ...
}
```

`const` vs. Function Parameters

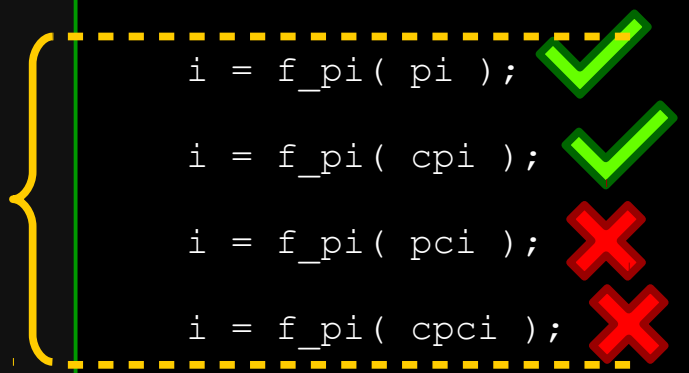
mainfconst3.c

```
int f_pi( int *a )
{
    return (*a);
}
...
```

`a` is a pointer to `int`.

mainfconst3.c

```
...
int i = 10;
const int ci = 20;
int *pi;
const int *pci;

int main( void )
{
    int *const cpi = &i;
    const int *const cpci = &i;

    i = f_pi( pi );
    i = f_pi( cpi );
    i = f_pi( pci );
    i = f_pi( cpci );
    ...
}
```



`f()` receives copies of the addresses of the integers that are being pointed.

If, through the pointer, a `const` value is going to be reinterpreted as non-const, the compiler raises an error.

`const` vs. Function Parameters

mainfconst4.c

```
int f_pci( const int *a )
{
    return (*a);
}
...
```

a is a pointer to a `const int`.

Pointer to `const` parameter prevents the modification of the pointed value.

`f()` receives copies of the addresses of the integers that are being pointed.

There won't be any problem if, through the pointer, a non-const value is reinterpreted as `const`.

mainfconst4.c

```
...
int i = 10;
const int ci = 20;
int *pi;
const int *pci;

int main( void )
{
    int *const cpi = &i;
    const int *const cpci = &i;

    i = f_pci( pi );
    i = f_pci( cpi );
    i = f_pci( pci );
    i = f_pci( cpci );
    ...
}
```

`const` vs. Function Parameters

mainfconst5.c

```
int f_cpi( int *const a )
{
    return (*a);
}
...
```

a is a `const` pointer to an `int`.

Why to pass a **`const`** pointer if **`f()`** will receive a copy of it (i.e. there is no risk of **`f()`** altering the original pointer)?

Restrictions via syntax is also an argument for the use of **`const`**.

`f()` receives copies of the addresses of the integers that are being pointed.

The copy of the pointer will be `const` within `f()`.

Through the pointer, the pointed value will be Interpreted as non-const!

mainfconst5.c

```
...
int i = 10;
const int ci = 20;
int *pi;
const int *pci;

int main( void )
{
    int *const cpi = &i;
    const int *const cpci = &i;

    i = f_cpi( pi );
    i = f_cpi( cpi );
    i = f_cpi( pci );
    i = f_cpi( cpci );
    ...
}
```

`const` vs. Function Parameters

mainfconst6.c

```
int f_cpci( const int *const a )
{
    return (*a);
}
...
```

a is a `const` pointer to a `const int`.

mainfconst6.c

```
...
int i = 10;
const int ci = 20;
int *pi;
const int *pci;

int main( void )
{
    int *const cpi = &i;
    const int *const cpci = &i;

    i = f_cpci( pi );
    i = f_cpci( cpi );
    i = f_cpci( pci );
    i = f_cpci( cpci );
    ...
}
```

`f()` receives copies of the addresses of the integers that are being pointed.

The copy of the pointer will be `const` within `f()`.

Through the pointer, the pointed value will be Interpreted as `const`!

`volatile` type Qualifier

*An object that has **volatile-qualified type may be modified** in ways **unknown** to the **implementation** or have other unknown side effects.*

C99 Spec.

`volatile` type Qualifier

- Example (from the C99 Spec.):

```
extern const volatile int real_time_clock;
```

- A good text about **`volatile`**:
  - How to Use C's **`volatile`** Keyword, by Nigel Jones.
    - http://www.barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword

Reducing the Cost of Function Calls

inline1.c

```
int F1( int a )
{
    return a + 1;
}

int main( int argc, char *argv[]
)
{
    int x;

    if ( argc > 1 )
        x = 2;
    else
        x = 1;

    x = F1( x );

    return x;
}
```

inline1.s

```
    .file   "main.c"
    .text
    .globl  F1
    .type   F1, @function
F1:
    pushq   %rbp
    movq %rsp, %rbp
    movl %edi, -4(%rbp)
    movl -4(%rbp), %eax
    addl $1, %eax
    popq %rbp
    ret
    ...
main:
    ...
    call F1
    ...
    ret
    ...
```

Function call!

Optimization OFF!

```
~$ gcc -Wall -Werror -pedantic -O0 -std=c99 -E inline1.c > inline1.i
~$ gcc -Wall -Werror -pedantic -O0 -std=c99 -fno-asynchronous-unwind-tables -S inline1.i
```

Reducing the Cost of Function Calls

inline1.c

```
int F1( int a )
{
    return a + 1;
}

int main( int argc, char *argv[]
)
{
    int x;

    if ( argc > 1 )
        x = 2;
    else
        x = 1;

    x = F1( x );

    return x;
}
```

The **compiler** itself may **decide** to **inline** some functions!

inline1.s

```
    .file   "main.c"
    .text
    .p2align 4,,15
    .globl  F1
    .type   F1, @function
F1:
    leal    1(%rdi), %eax
    ret
    .size   F1, .-F1
    .section .text.startup,"ax",@progbits
    .p2align 4,,15
    .globl  main
    .type   main, @function
main:
    xorl    %eax, %eax
    cmpl    $2, %edi
    setge   %al
    addl    $2, %eax
    ret
```

No function call! `F1()` was inlined!

Optimization ON!

```
~$ gcc -Wall -Werror -pedantic -O2 -std=c99 -E inline1.c > inline1.i
~$ gcc -Wall -Werror -pedantic -O2 -std=c99 -fno-asynchronous-unwind-tables -S inline1.i
```

Reducing the Cost of Function Calls

inline2.c

```
void InsertionSort( int *v, int size ) {
    int j;
    int key;

    for( int i = 1; i < size; i++ ) {
        key = v[ i ];
        j = i - 1;

        while( ( j >= 0 ) && ( v[j] > key ) ) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}

int y[5] = { 5, 4, 3, 2, 1 };

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    return y[0];
}
```

inline2.s

```
        ...
        .globl    InsertionSort
        .type     InsertionSort, @function
InsertionSort:
        pushq     %rbp
        movq %rsp, %rbp
        ...
        ret
        .size     InsertionSort, .-InsertionSort
        .globl    y
        .data
        .align 16
        .type     y, @object
        .size     y, 20
y:
        .long     5
        ...
main:
        ...
        call InsertionSort
        ...
        ret
        ...
```

Function call!

Optimization OFF!

```
~$ gcc -Wall -Werror -pedantic -O0 -std=c99 -E inline2.c > inline2.i
~$ gcc -Wall -Werror -pedantic -O0 -std=c99 -fno-asynchronous-unwind-tables -S inline2.i
```

Reducing the Cost of Function Calls

inline2.c

```
void InsertionSort( int *v, int size ) {
    int j;
    int key;

    for( int i = 1; i < size; i++ ) {
        key = v[ i ];
        j = i - 1;

        while( ( j >= 0 ) && ( v[j] > key ) ) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}

int y[5] = { 5, 4, 3, 2, 1 };

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    return y[0];
}
```

In this case, the function call **was not optimized** by the compiler.

inline2.s

```
        ...
        .globl    InsertionSort
        .type     InsertionSort, @function
InsertionSort:
        cmpl $1, %esi
        jle  .L1
        ...
        rep ret
        ...
        .size     InsertionSort, .-InsertionSort
        .section  .text.startup,"ax",@progbits
        .p2align 4,,15
        .globl    main
        .type     main, @function
main:
        ...
        call InsertionSort
        ...
        ret
        ...
```

Function call!

How do we **ask** the **compiler** to **inline functions**?

Optimization ON!

```
~$ gcc -Wall -Werror -pedantic -O2 -std=c99 -E inline2.c > inline2.i
~$ gcc -Wall -Werror -pedantic -O2 -std=c99 -fno-asynchronous-unwind-tables -S inline2.i
```

`inline` Function Specifier

*A function **declared** with an **inline** function specifier is an **inline function**. (...) Making a function an inline function suggests that calls to the function be **as fast as possible**. The extent to which such suggestions are effective is **implementation-defined**.*

C99 Spec

Simple `inline` Example

inline2.c

```
inline void InsertionSort( int *v, int size ) {
    int j;
    int key;

    for( int i = 1; i < size; i++ ) {
        key = v[ i ];
        j = i - 1;

        while( ( j >= 0 ) && ( v[j] > key ) ) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}

int y[5] = { 5, 4, 3, 2, 1 };

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    return y[0];
}
```

inline2.s

No 'outline' code for `F1()` was generated!

```
    .file   "main.c"
    .section .text.startup,"ax",@progbits
    .p2align 4,,15
    .globl  main
    .type   main, @function
main:
    xorl %esi, %esi
.L6:
    movl y+4(,%rsi,4), %ecx
    ...
    movl y(%rip), %eax
    ret
    ...
```

No function call! `F1()` was inlined!

Optimization ON!

```
~$ gcc -Wall -Werror -pedantic -O2 -std=c99 -E inline2.c > inline2.i
~$ gcc -Wall -Werror -pedantic -O2 -std=c99 -fno-asynchronous-unwind-tables -S inline2.i
```

Splitting the Code

inline2.c

```
inline void InsertionSort( int *v, int size ) {
    int j;
    int key;

    for( int i = 1; i < size; i++ ) {
        key = v[ i ];
        j = i - 1;

        while( ( j >= 0 ) && ( v[j] > key ) ) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}
```

```
int y[5] = { 5, 4, 3, 2, 1 };

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    return y[0];
}
```

insertion.c

```
inline void InsertionSort( int *v, int size ) {
    int j;
    int key;

    for( int i = 1; i < size; i++ ) {
        key = v[ i ];
        j = i - 1;

        while( ( j >= 0 ) && ( v[j] > key ) ) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}
```

inline3.c

Add declaration!

```
inline void InsertionSort( int *v, int size );

int y[5] = { 5, 4, 3, 2, 1 };

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    return y[0];
}
```

Splitting the Code

insertion.c

```
inline void InsertionSort( int *v, int size ) {
    int j;
    int key;

    for( int i = 1; i < size; i++ ) {
        key = v[ i ];
        j = i - 1;

        while( ( j >= 0 ) && ( v[j] > key ) ) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}
```

inline3.c

```
inline void InsertionSort( int *v, int size );

int y[5] = { 5, 4, 3, 2, 1 };

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    return y[0];
}
```

```
-Wall -Werror -pedantic -O2 -std=c99
```

```
~$ gcc ... -E insertion.c > insertion.i
~$ gcc ... -fno-asynchronous-unwind-tables -S insertion.i
~$ gcc ... -c insertion.s
~$ gcc ... -E inline3.c > inline3.i
~$ gcc ... -fno-asynchronous-unwind-tables -S inline3.i

main.c:3:13: error: inline function ‘InsertionSort’
declared but never defined [-Werror]
 inline void InsertionSort( int *v, int size );
             ^
main.c:3:13: error: inline function ‘InsertionSort’
declared but never defined [-Werror]
cc1: all warnings being treated as errors
```

insertion.s

```
        .file "insertion.c"
        .ident      "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4"
        .section    .note.GNU-stack,"",@progbits
```

The **definition** of an **inline** function must be placed in the **same translation unit** where it will be **inlined**!

How to keep an **inline function definition** in a **separate** file?

Splitting the Code

insertion.c (old)

```
inline void InsertionSort( int *v, int size ) {
    int j;
    int key;

    for( int i = 1; i < size; i++ ) {
        key = v[ i ];
        j = i - 1;

        while( ( j >= 0 ) && ( v[j] > key ) ) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}
```

insertion.h (new)

```
inline void InsertionSort( int *v, int size ) {
    int j;
    int key;

    for( int i = 1; i < size; i++ ) {
        key = v[ i ];
        j = i - 1;

        while( ( j >= 0 ) && ( v[j] > key ) ) {
            v[j + 1] = v[j];
            j = j - 1;
        }
        v[j + 1] = key;
    }
}
```

inline3.c (old)

```
inline void InsertionSort( int *v, int size );

int y[5] = { 5, 4, 3, 2, 1 };

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    return y[0];
}
```

inline3.c (new)

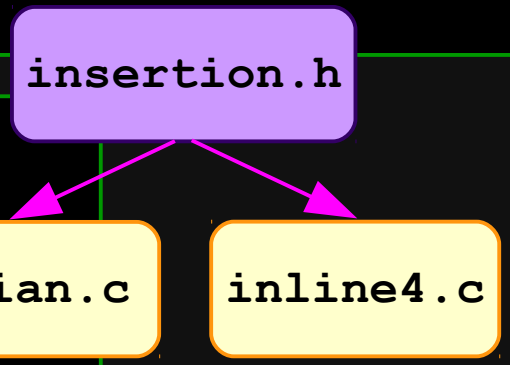
```
#include "insertion.h"

int y[5] = { 5, 4, 3, 2, 1 };

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    return y[0];
}
```

Is it possible for a **inline** definition to be **included** in **multiple files**?

Inline Function in Multiple Files

insertion.h

```
inline void InsertionSort( int *v, int size )
{ ... }
```



median.c

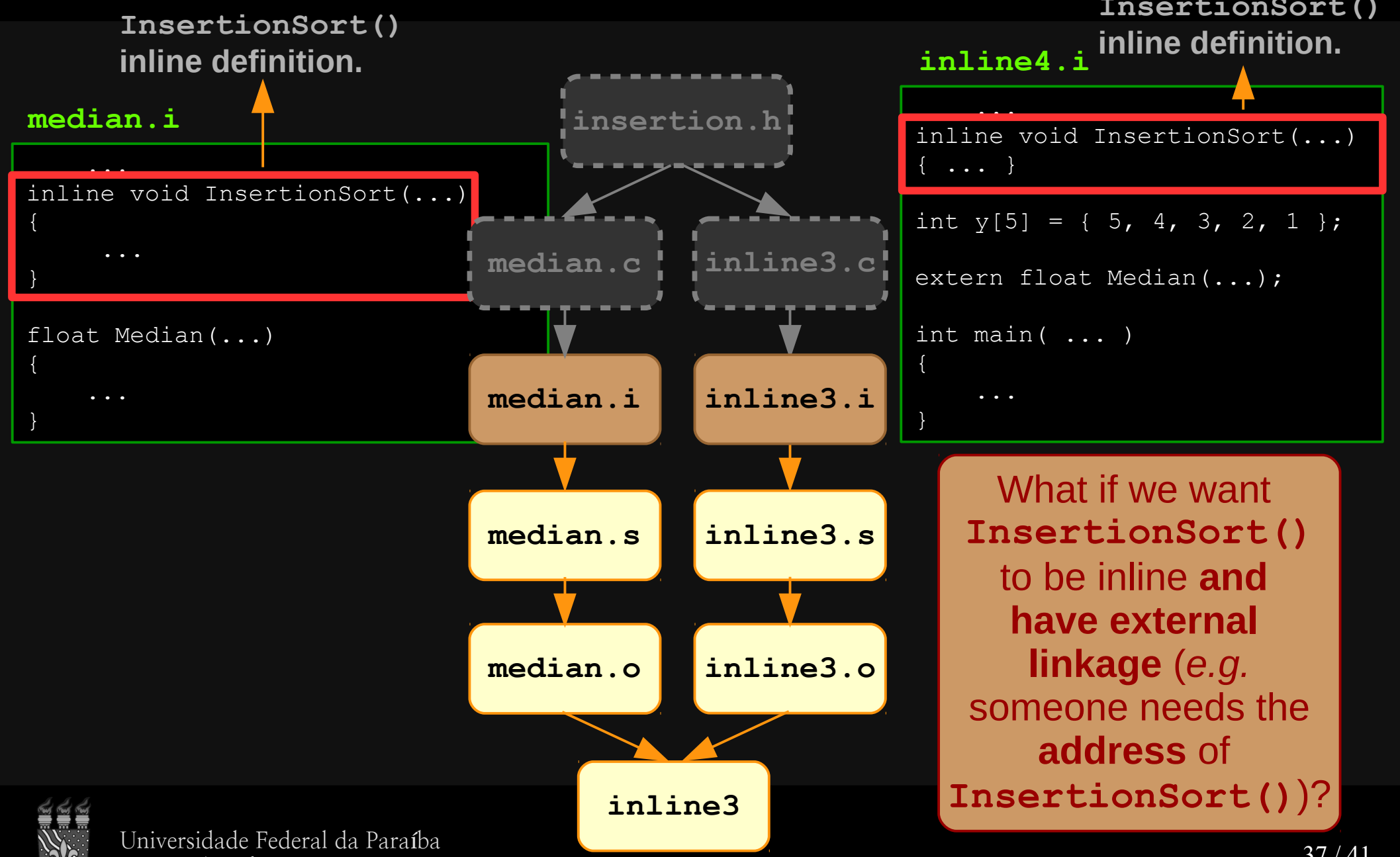
```
#include <string.h>
#include <stdlib.h>
#include "insertion.h"

float Median( int *v, int size )
{
 int *tmp = malloc(sizeof(int) * size);
 memcpy(tmp, v, sizeof(int) * size);

 InsertionSort(tmp, size);

 float median;

 if (size%2 == 0)
  median=(tmp[size/2]+tmp[size/2-1])*0.5f;
 else
  median = tmp[(int) size/2];

 free(tmp);

 return median;
}
```

inline4.c

```
#include “insertion.h”

int y[5] = { 5, 4, 3, 2, 1 };

extern float Median(int *v, int size);

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    float m = Median( y, 5 );
    return ( int ) m  + y[0];
}
```

```
~$ gcc -Wall -Werror -pedantic -O2 -std=c99 -E median.c > median.i
~$ gcc -Wall -Werror -pedantic -O2 -std=c99 -E inline4.c > inline4.i
```

Inline Function in Multiple Files



What if we want **`InsertionSort()`** to be inline **and have external linkage** (*e.g.* someone needs the **address** of **`InsertionSort()`**)?

Inline Function with External Linkage

- Suppose that **`Median()`** needs to call **`InsertionSort()`** through its address.
- Two possible ways are:
  - Redefining **`InsertionSort()`** as non-inline.

    ... which is not cool because implies in duplicate code that will have to be maintained consistent manually.
  - Redeclaring **`InsertionSort()`** with **`extern`**.

    ... which is nicer, since the duplicated code will be kept consistent automatically!

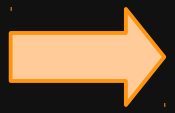
Inline Function with External Linkage

insertion.h

```
inline void InsertionSort( ... )
{ ... }
```

median.c (old)

```
#include <string.h>
#include <stdlib.h>
#include "insertion.h"

float Median( ... ) {
 ...
 InsertionSort(tmp, size);
 ...
}
```

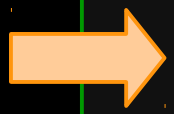


median.c (new)

```
#include <string.h>
#include <stdlib.h>

extern void InsertionSort( ... );

float Median( ... ) {
 ...
 InsertionSort(tmp, size);
 ...
}
```

inline4.c (old)

```
#include “insertion.h”

int y[5] = { 5, 4, 3, 2, 1 };

extern float Median( ... );
int main( int argc, char *argv[] ) {
    ...
}
```



inline4.c (new)

```
#include “insertion.h”

int y[5] = { 5, 4, 3, 2, 1 };

extern inline void InsertionSort( ... );

extern float Median(int *v, int size);
int main( int argc, char *argv[] ) {
    ...
}
```

Forces the inclusion of an 'outline' definition of `InsertionSort()`.

Inspect the **intermediary code** generated!

Inline Function with External Linkage

insertion.h

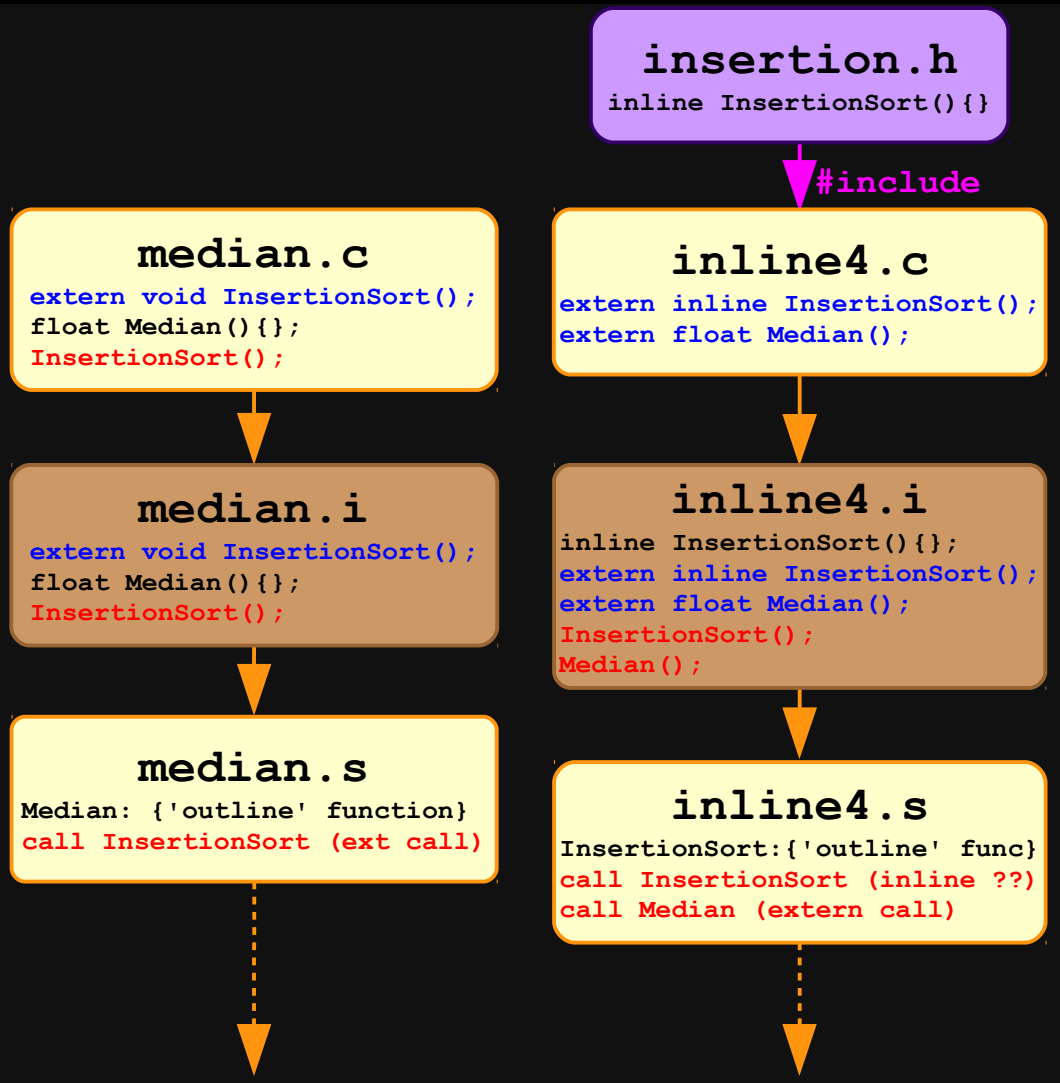
```
inline void InsertionSort( int *v, int size )
{ ... }
```

median.c

```
#include <string.h>
#include <stdlib.h>

extern void InsertionSort( int *v, int size );

float Median( int *v, int size )
{
 ...
 InsertionSort(tmp, size);
 ...
}
```

inline4.c

```
#include "insertion.h"

int y[5] = { 5, 4, 3, 2, 1 };

extern inline void InsertionSort( int *v, int size );

extern float Median(int *v, int size);

int main( int argc, char *argv[] ) {
    InsertionSort( y, 5 );
    float m = Median( y, 5 );
    return y[0];
}
```



Exercise: `inline/const` vs. Macros

Compare both codes below:

Can we **instruct** the **compiler** to generate the (at least approximately) **same low level code** by just **substituting macros** by `const` and `inline`?

nomacro.c

```
int k = 10;

int OffsetFunction( int w ) { return w + k; }

int main( int argc, char *argv[] )
{
    int x;

    if ( argc > 1)
        x = 2;
    else
        x = 1;

    int y = 3 * OffsetFunction( x );

    return y;
}
```

macro.c

```
#define OFFSET_VALUE 10

#define OFFSET_FUNCTION( val ) ( val + OFFSET_VALUE )

int main( int argc, char *argv[] )
{
    int x;

    if ( argc > 1)
        x = 2;
    else
        x = 1;

    int y = 3 * OFFSET_FUNCTION( x );

    return y;
}
```