SYSTEM SOFTWARE

Part 1: C = Language + Environment

Part 2: Linux Programming Interface

Course Material

- Slides & code on Toledo
- Linux man/info-pages!!
- References to online material: see slides
  - E.g.: “The C Programming Language”, Kernighan, Ritchie
    - publications.gbdirect.co.uk/c_book/
  - E.g. GNU C Reference Manual
    - www.gnu.org/software/gnu-c-manual/gnu-c-manual
- Do 'interactive C programming'
    - http://www.learn-c.org
    - http://fresh2refresh.com/c-tutorial-for-beginners/
    - http://www.tutorialspoint.com/cprogramming/c_useful_resources.htm

<u>HIGHLY RECOMMENDED</u>

subscribe to the newsletter of www.cprogramming.com!

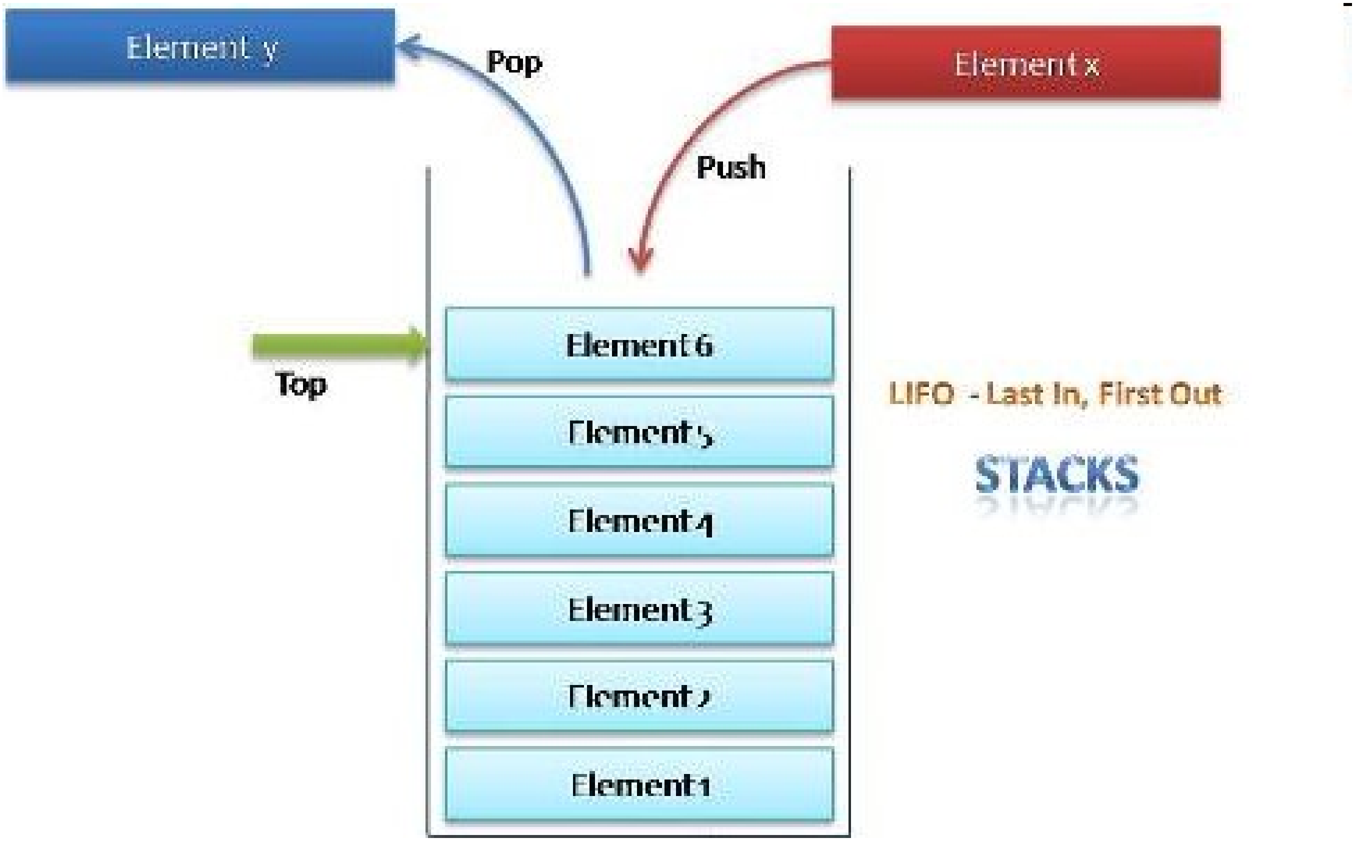
Course Material

- Book references
  - Beginner
    - C for Programmers (with an introduction to C11) (P. Deitel, H. Deitel)
    - Understanding and Using C Pointers (R. Reese)
    - C Pocket Reference (P. Prinz, U. Kirch-Prinz)
  - Intermediate
    - 21st Century C: C Tips from the New School (B. Klemens)
    - Learn c the Hard Way (Zed A. Shaw)
    - Intermediate C Programming (Yung-Hsiang Lu)
    - Mastering Algorithms with C (K. Loudon)
    - C Interfaces and Implementations: Techniques for Creating Reusable Software (D. R. Hanson)
  - Advanced
    - Expert C Programming – Deep C Secrets (P. Van Der Linden)

LECTURE C1

The basics of C programming

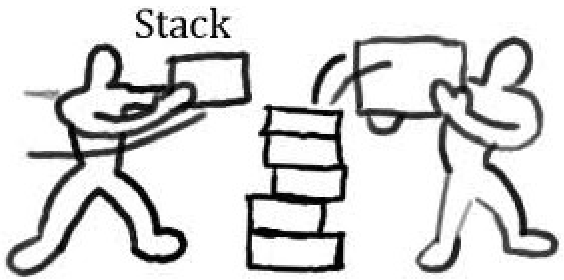
A Stack Example



A Stack Example

- Study the code …

[Toledo: StackV1]



- Build and run the code
  - Gcc -Wall main.c
    - ./a.out
  - Gcc main.c –o prog.exe
    - ./prog.exe

C Is Not Object-Oriented!

- Data is accessible via variables (local or global) and modified in functions
  - There are no classes nor objects
    - No constructor: you need to manage memory!
    - No garbage collector: you need to clean up!
    - No inheritance
    - No polymorfism
  - Access to data / functions cannot be controlled as public, private or protected

C Is Not Object-Oriented!

- C program is a collection of functions calling each other
  - Functions cannot be ‘nested’!
    - Use function prototypes to avoid dependency problems
  - Program starts with ‘main’ function

C is a 'medium-high' programming language!

Assembler < C < Java, Python,...

The Standard Library

- Collection of functions defined by ANSI but not part of the C language

- Functions are grouped in ‘header’ files
  - E.g. math.h, string.h, time.h, …

- Usage: #include<*libname.h*>

The Standard Library

- A few references to the standard library & example code
  - http://cplusplus.com/reference/clibrary/
    - www.java2s.com/Code/C/CatalogC.htm
  - Reference with full details
    - www.gnu.org/software/libc/manual

Prerequisites Lecture 2 (**Homework!**)

- Master the C programming basics
  - C data types
    - Atomic: int, char, float, double, void
    - Type qualifiers: short, long, long long, unsigned, signed
    - Structured: array, enum, struct
  - Variables
  - Arithmetic expressions
    - Typecasting, e.g. convert int x to float: (float)x
  - Selection
    - If, if else, switch, conditional statement
  - Looping
    - Do-loop, while-loop, for-loop
    - Break and continue
  - Functions
    - Arguments / return values / local and global variables
  - Standard library basics

Prerequisites Lecture 2 (**Homework!**)

- Suggestions
  - Read a C programming tutorial
    - E.g. On Toledo “Starting guide to C programming”
    - E.g. publications.gbdirect.co.uk/c_book/
      - Chapter 1, 2, 3, 4 (section 4.1, 4.2, 4.3), 5 (section 5.1, 5.2), 6 (section 6.2 but not 6.2.1, 6.5)
      - http://en.wikibooks.org/wiki/C_Programming

Prerequisites Lecture 2 (**Homework!**)

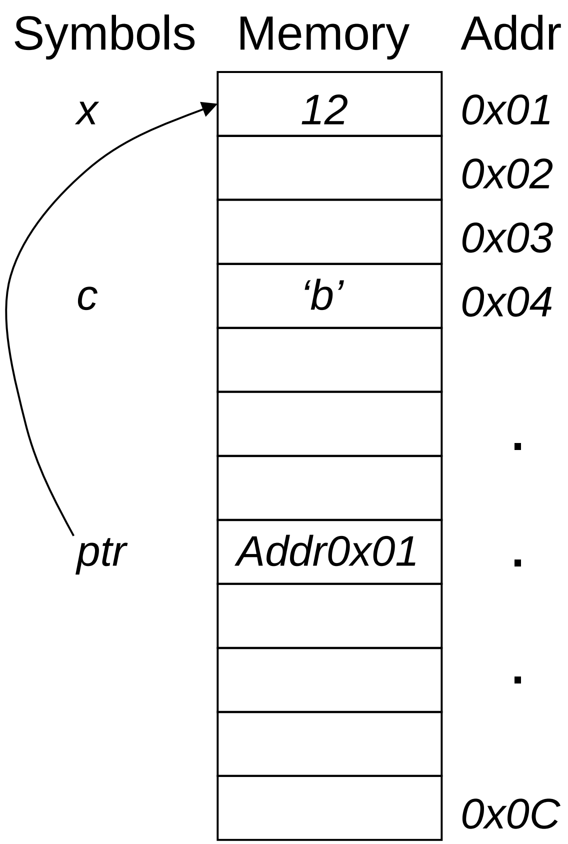
- Even better: program simple C code
  - Your Linux environment is up and running
    - Use some editor (kate, gedit, geany, vim, …)
    - Compile in a terminal with ‘gcc’
  - Your Linux is not ok
    - Do 'online programming'
      - http://www.learn-c.org
      - http://fresh2refresh.com/c-tutorial-for-beginners/
      - http://www.tutorialspoint.com/cprogramming/c_useful_resources.htm
      - https://ideone.com/
      - http://www.codingame.com : game-style c programming

LECTURE C2

Pointers and Dynamic Memory

What Are Pointers in C?

- Pointer = variable that contains a memory address
  - Data type: ‘address’
- A pointer often contains the address of another variable
  - Pointer ‘points’ or ‘references’ to that variable
- Example
  - int x; x = 12;
  - char c;   c = ‘b’;
  - int *ptr;   ptr = &x;
  - Printf(“%d”, *ptr);



What Are Pointers in C?

- &-operator
  - Used to obtain the address of a variable
  - E.g. char c;

    &c is the address of the memory allocation that stores c
- *-operator
  - Used to declare a pointer
  - E.g. char *cptr;
  - cptr is a pointer to a memory location that can contain a char
- *-operator
  - Also used to dereference a pointer, i.e. to access the variable to which the pointer references
  - E.g. *cptr = ‘b’;
- NULL is a special ‘address’ (0x0000) to initialize pointers
  - E.g. float *p = NULL; ==> *p is NOT allowed!

What Are Pointers in C?

- Example and graphical representation

```
int x, y;
int *p, *q = NULL;
```


```
x = 1; y = 0;
p = &x;  // *p is 1
```

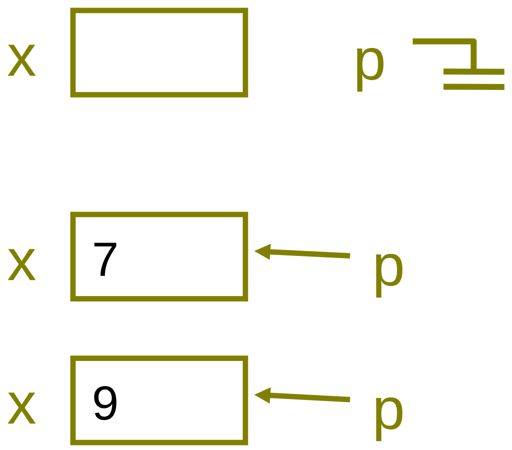

```
y = *p + x;
*p = 0;
```


```
q = p;
```

What Are Pointers in C?

- Example: using pointers to change variables

```
int x;
int *p = NULL;

x = 7;
p = &x;
*p = 9;

printf("value of x = %d", x);
```



Pointers Quiz!

- Possible or not possible? Draw a memory layout!

```
Int x;
Int *p = NULL;
Int *q = &x;
```

1. x = NULL;
2. p = x;
3. x = *p;
4. *q = 7;
5. *q = &x;
6. q = p;
7. &x = q;

Pointers to pointers

- Pointers to all kinds of data types
  - Pointers to int, float, char, double, …
  - Pointers to structs, ...
- Pointers to pointers? Oh YES!

```
int ***p;
int **q;
int *r;
int x;

r = &x; q = &r; p = &q;
```

Pointers

- A nice explanation on pointers and memory drawings can be found in the following paper (Toledo)
  - “Pointers and Memory”, section 1, Nick Parlante, Standford University

Why Pointers?

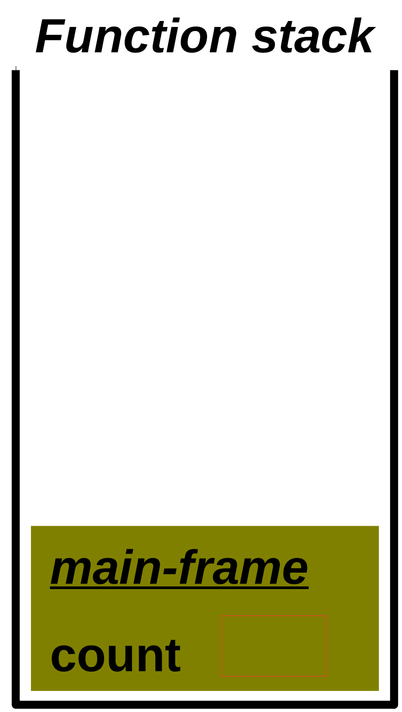
- Pointers allow flexible manipulation of data and code
  - Pointers and function parameter passing
  - Pointers and arrays
    - Static arrays, dynamic arrays (dynamic memory)
  - Pointers and dynamic memory / structures
    - Linked lists, trees, heaps, …
  - Pointers and functions

See later!

Pointers and Parameter Passing

- The function (/application/system) stack
  - C requires a 'stack' for the evaluation of functions

```
void add( int x )
{
    int total = 10;
    x += total;
}

void main()
{
    int count = 0;
    add( count );
    printf(“Count = %d“,
        count);
}
```

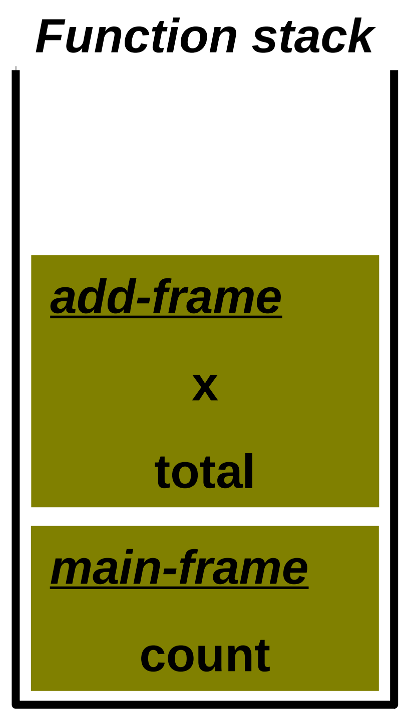


A function call results in a new frame on the function stack
- parameters
- local vars
- NO global vars!

Pointers and Parameter Passing

- The function (/application/system) stack
  - C requires a 'stack' for the evaluation of functions

```
void add( int x )
{
    int total = 10;
    x += total;
}

void main()
{
    int count = 0;
    add( count );
    printf(“Count = %d“,
        count);
}
```

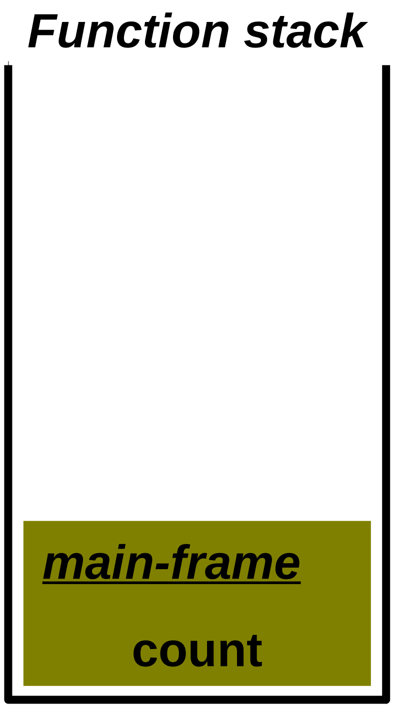



```
A function call results
in a new frame on the
function stack
- parameters
- local vars
- return address!
- NO global vars!
```

Pointers and Parameter Passing

- The function (/application/system) stack
  - C requires a 'stack' for the evaluation of functions

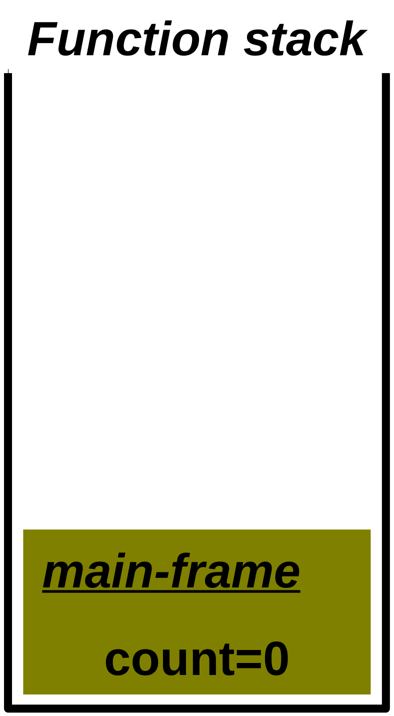
```
void add( int x )
{
    int total = 10;
    x += total;
}

void main()
{
    int count = 0;
    add( count );
    printf(“Count = %d“,
        count);
}
```



Pointers and Parameter Passing

- The function (/application/system) stack
  - Call-by-value parameter passing

```
    void add( int x )
    {
        int total = 10;
        x += total;
    }

    void main()
PC  {
        int count = 0;
        add( count );
        printf(“Count = %d“,
             count);
    }
```

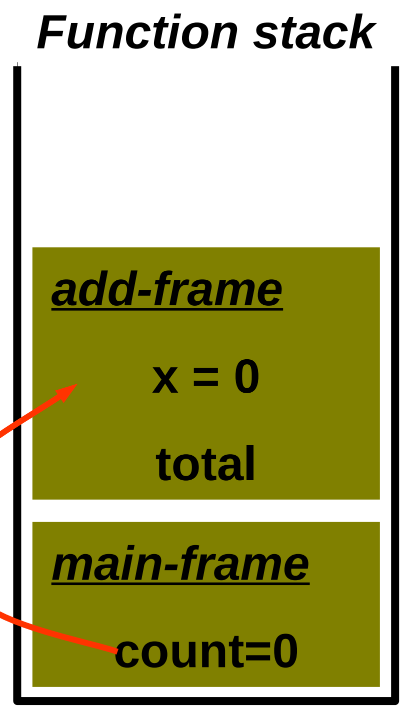


Pointers and Parameter Passing

```
void add( int x )
{
    int total = 10;
    x += total;
}

void main()
{
    int count = 0;
    add( count );
    printf(“Count = %d“,
         count);
}
```

PC

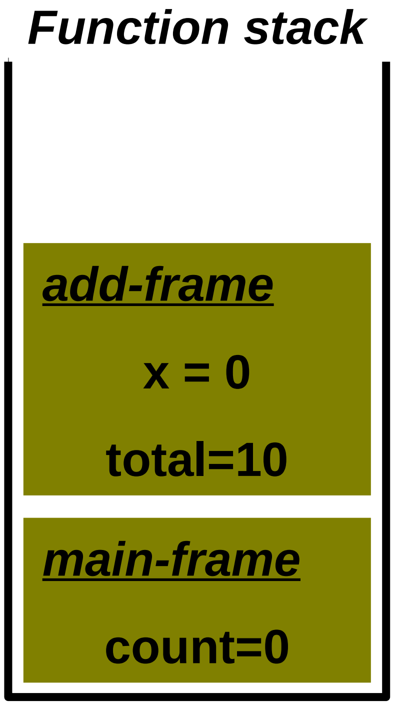
Copy value of count to x!



Pointers and Parameter Passing

PC

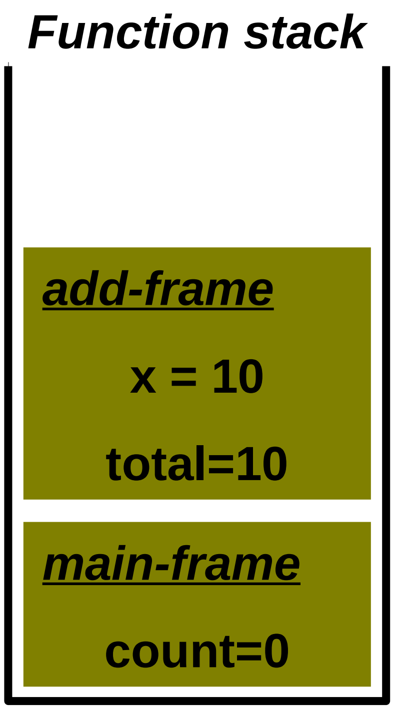
```
void add( int x )
{
    int total = 10;
    x += total;
}

void main()
{
    int count = 0;
    add( count );
    printf("Count = %d",
        count);
}
```



Pointers and Parameter Passing

PC

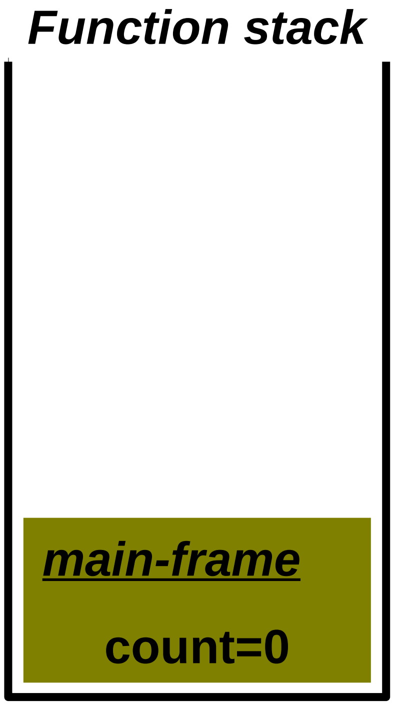
```
void add( int x )
{
    int total = 10;
    x += total;
}

void main()
{
    int count = 0;
    add( count );
    printf(“Count = %d“,
         count);
}
```



Pointers and Parameter Passing

```
void add( int x )
{
   int total = 10;
   x += total;
}

void main()
{
   int count = 0;
   add( count );
   printf(“Count = %d“,
       count);
}
```

PC

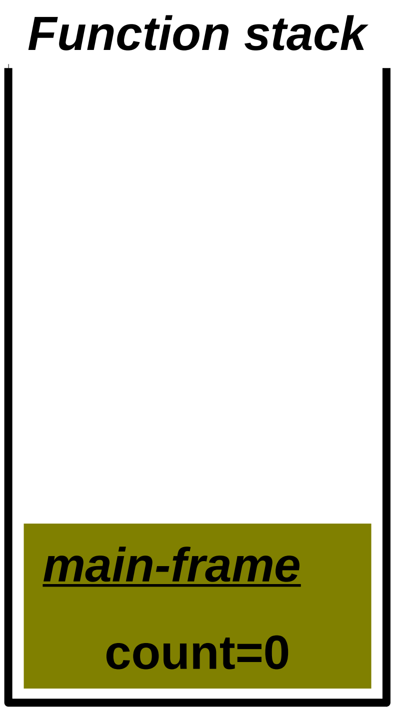


Pointers and Parameter Passing

- The function (/application/system) stack
  - Call-by-reference parameter passing

```
void add( int *x )
{
    int total = 10;
    *x += total;
    x = &total;
    (*x)++;
}

void main() {
    int count = 0;
    add( &count );
    printf("Count = %d",
        count);
}
```

PC

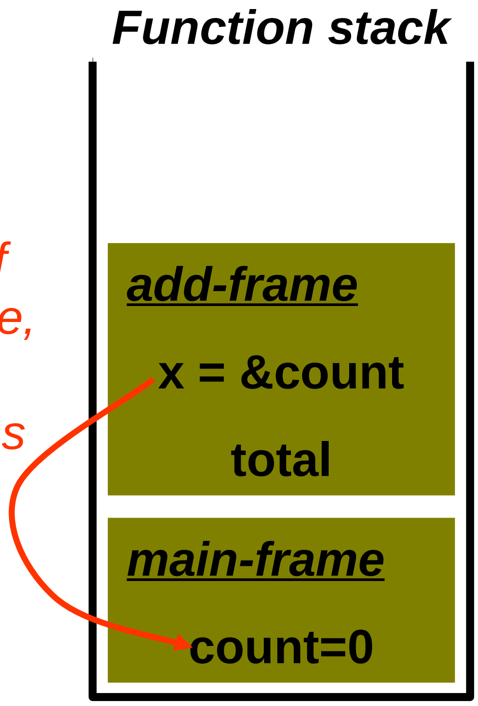


Pointers and Parameter Passing

```
void add( int *x )
{
    int total = 10;
    *x += total;
    x = &total;
    (*x)++;
}

void main() {
    int count = 0;
    add( &count );
    printf("Count = %d",
        count);
}
```

PC

Copy address of count to x, hence, x points to the same memory as 'count'!

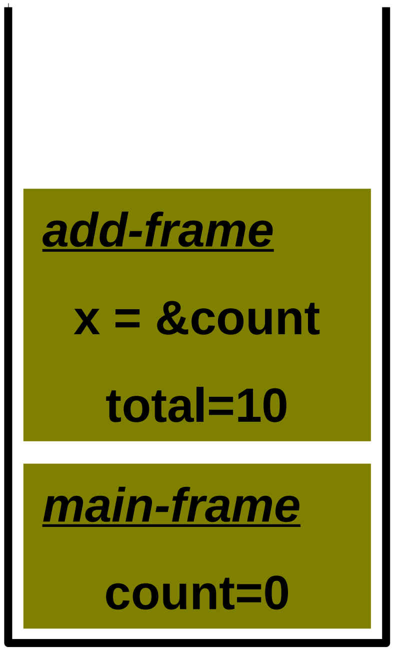


Pointers and Parameter Passing

PC

```
void add( int *x )
{
    int total = 10;
    *x += total;
    x = &total;
    (*x)++;
}

void main() {
    int count = 0;
    add( &count );
    printf("Count = %d",
        count);
}
```

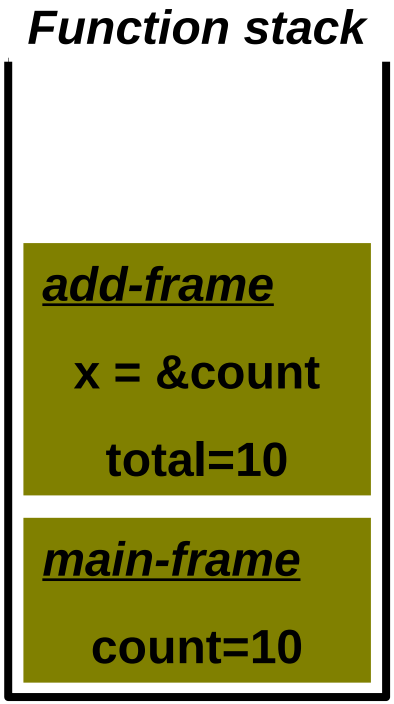
Function stack



Pointers and Parameter Passing

PC

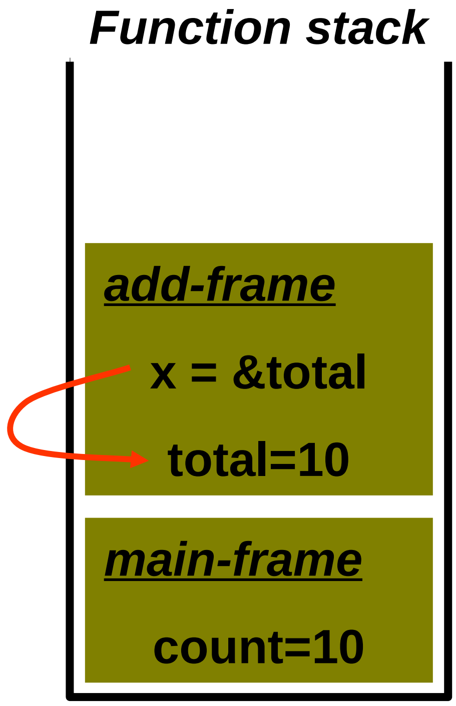
```
void add( int *x )
{
    int total = 10;
    *x += total;
    x = &total;
    (*x)++;
}

void main() {
    int count = 0;
    add( &count );
    printf(“Count = %d“,
        count);
}
```



Pointers and Parameter Passing

PC

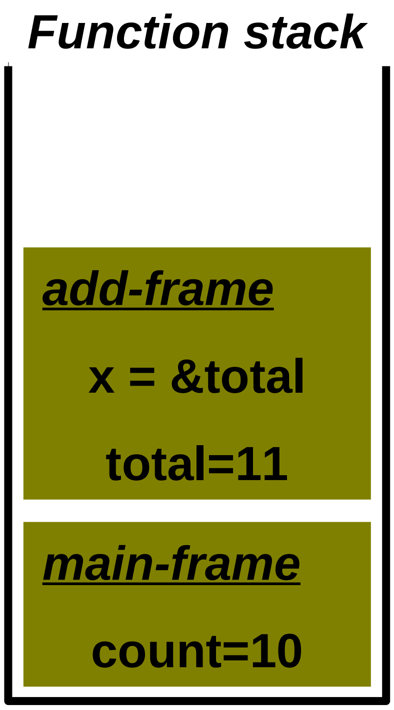
```
void add( int *x )
{
    int total = 10;
    *x += total;
    x = &total;
    (*x)++;
}

void main() {
    int count = 0;
    add( &count );
    printf(“Count = %d“,
        count);
}
```



Pointers and Parameter Passing

PC

```
void add( int *x )
{
    int total = 10;
    *x += total;
    x = &total;
    (*x)++;
}

void main() {
    int count = 0;
    add( &count );
    printf(“Count = %d“,
        count);
}
```

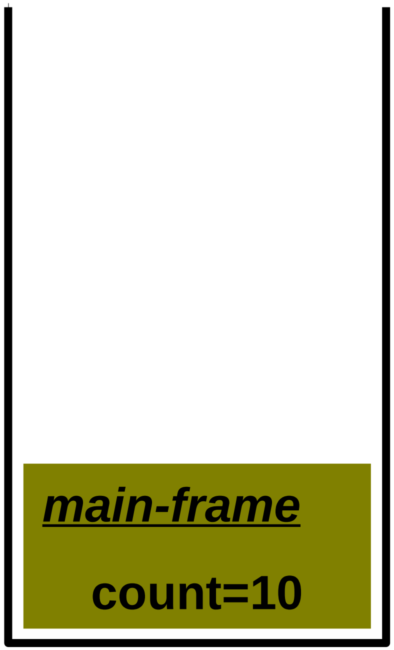


Pointers and Parameter Passing

```
void add( int *x )
{
    int total = 10;
    *x += total;
    x = &total;
    (*x)++;
}

void main() {
    int count = 0;
    add( &count );
    printf(“Count = %d“,
        count);
}
```

PC

Function stack



Pointers and Parameter Passing

- A nice explanation on the stack and parameter passing can be found in the following paper (Toledo)
  - "Pointers and Memory", section 2 and 3, Nick Parlante, Standford University

Const and Pointers

- Const-keyword defines constants
  - E.g. const int x = 3;
  - Or: int const x = 3;

- Const and pointers
  - const int * x; ➔ variable ptr to a constant int
  - Example
    - int a = 0;
    - const int * x = &a;
    - *x = 10; => NOT allowed!
    - a = 10; => allowed - *x is now also changed

  - Remark:
    - int const *x; is the same as const int *x;

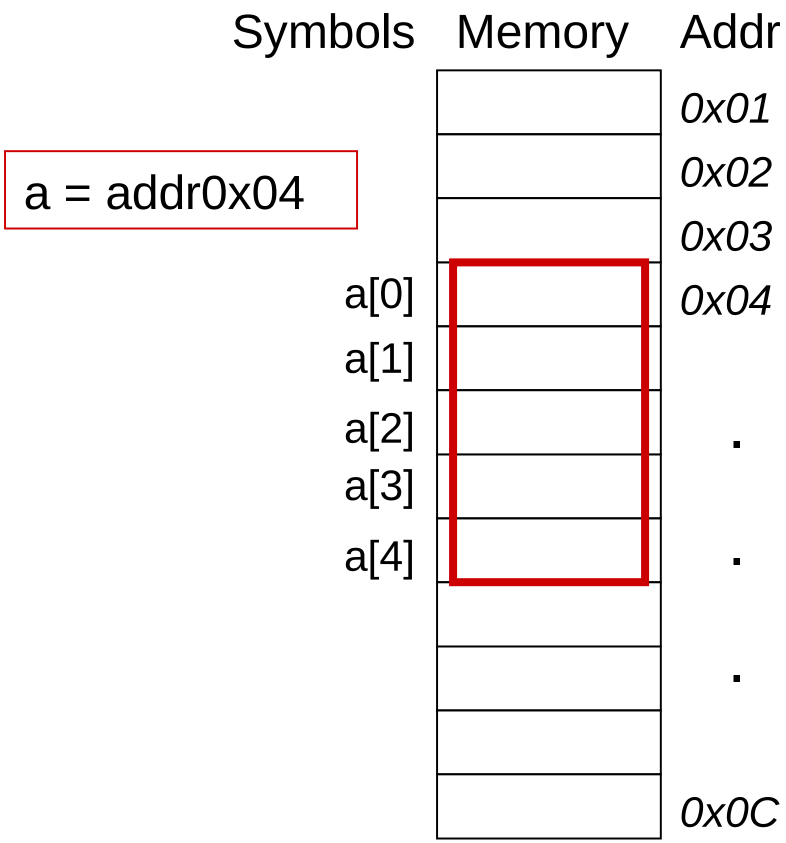
Const and Pointers

- Const and pointers
  - int * const x; ➔ constant ptr to a variable int
  - Example:
    - int a =0, b = 10;
    - int * const x = &a;
    - x = &b;   => NOT allowed!
    - *x = b;   => allowed – a is also changed

  - const int * const x;   ➔ constant ptr to a constant int

Pointers and Arrays

- int a[5];
  - a contains the address of a[0]
  - *a is the same as a[0]
  - BUT: a is NOT a pointer!



Pointers and Arrays

- ‘pointer’ arithmetic
  - int *p, *q;
    - p+3 : address of ‘3rd’ integer
    - *(p+3) : ‘3rd” integer
    - p[3] or *(p+3)
    - ++p equivalent to p+1
  - If p and q point into same array, p – q is number of elements between p and q

Pointers and Arrays

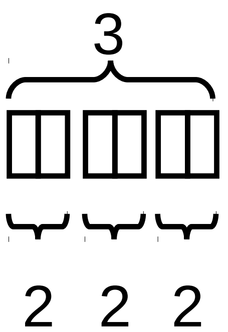
- Nice to know …
  - Initialize an array
    - Int a[] = {1, 2, 3, 4, 5 };
    - Int a[5] = {0};
    - Int a[5] = {[2]=100, [4]=200};

Pointers and Arrays

- Nice to know …
  - Typedef array
    - typedef int array_t[MAX];
  - Array of pointers
    - int * a[MAX];
  - Pointer to array
    - int (*a) [MAX];

Pointers and Arrays

- Multi-dimensional arrays and pointers
  - Example: int a[3][2];
    - a[i]   => int *a_ptr = (int *)a[i];
    - a[i][j]   => *(a_ptr + j)



  - Or: look at a as a 1-dim. array: int a_1d[3*2];
    - Int *a_ptr = (int *)a_1d;
    - a[i][j]   => a_1d[2*i+j]   => *(a_ptr+2*i + j)

Pointers and Arrays

- Example: command line arguments

> test.exe –g inp_file

The arguments to the program are:
1 = test.exe
2 = -g
3 = inp_file

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

argc = #args

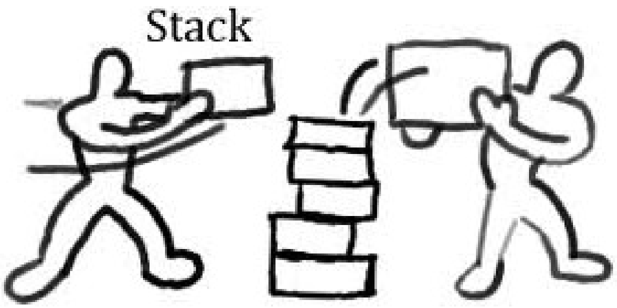
argv = pointer array to args

= NULL terminated array of strings

Pointers and Arrays

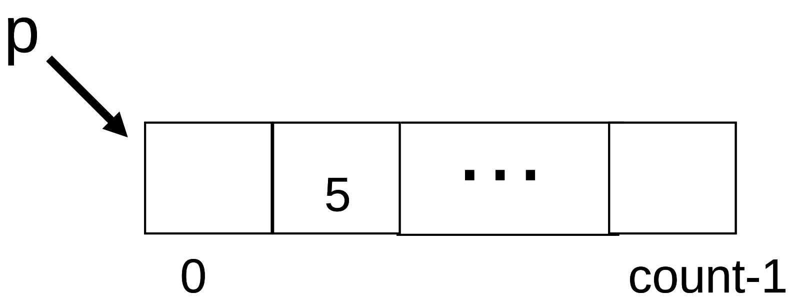
- Study the code …

[Toledo: cmdargs]



Dynamic Memory

- void *malloc(size_t size);
  - Allocates ‘size’ bytes of dynamic memory
  - Malloc returns a ‘void pointer’

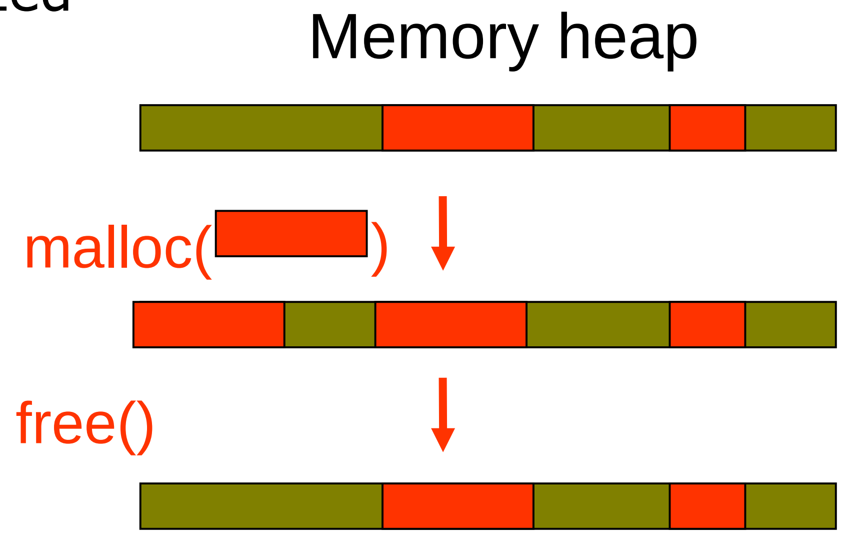


- Example

```
int count;
int *p;
scanf(“%d”, &count);
p = (int *) malloc(  count * sizeof(int) );
if ( p == NULL ) printf(“\nout of memory!”);
…
p[1] = 5;
…
free(p);
```

Dynamic Memory

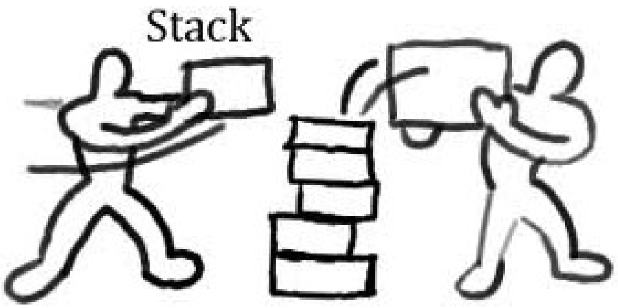
- The STL function ‘malloc’ is used for allocation of dynamic memory;
- The STL function ‘free’ is used for de-allocation of dynamic memory
- The STL function ‘realloc’ changes the memory size allocated to p to a new size
  - Additional memory is not initialized



Dynamic Memory

- Study the code …

[Toledo: StackV3]



Pointers and Arrays

- Safer printf() functions
  - int snprintf (str, n, *format, arg1, arg2, ..., argn*)
    - Same as sprintf() but at most n bytes (including \0) are printed to the string str
  - int asprintf (&str, *format, arg1, arg2, ..., argn*)
    - Same as sprintf() but allocates enough memory to contain the output (including \0)

Pointers and Parameter Passing

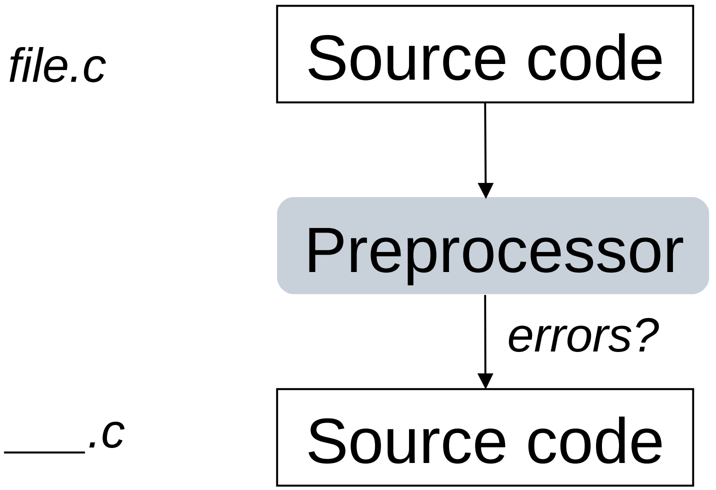
- Everything you need to know about pointers in C:
  - “Understanding and Using C Pointers”, Richard Reese, O’Reilly, ISBN: 978-1-449-34418-4

LECTURE C3

The Preprocessor

Preprocessor

- Tasks
  - Removes comments
  - Interprets pre-processor directives denoted by #
    - #define
    - #include



- 'gcc main.c' calls preprocessor automatically, but the output of the preprocessor can also be obtained with 'gcc –E main.c'

Conditional Compilation

- Preprocessor directives allow to ‘conditionally select’ code given to the compiler

```
#define LARGE
main()
{
   /* do something */
#ifdef LARGE
   int a[1000];
#else
   int a[100];
#endif
   /* do something */
}
```

Conditional Compilation

```
#ifndef LARGE
   /* compile this */
#else
   /* compile this */
#endif
```

```
#if defined(LARGE)
   /* compile this */
#else
   /* compile this */
#endif
```

```
#if CHOICE == 1
   /* compile this */
#elif CHOICE == 2
   /* compile this */
#elif CHOICE == 3
   /* compile this */
#else
   /* compile this */
#endif
```

Conditional Compilation

- How to define a preprocessor symbol?
  - #define *name*
  - #define *name value*
  - Compiler option
    - Gcc –D *name=value* …
- #undef *name*

Conditional Compilation

- Example: add debug code

```
#ifdef DEBUG
  printf("status ok!\n");
#endif

#undef DEBUG   // no debug info anymore

#ifdef DEBUG
  printf("status ok!\n");
#endif
```

```
> gcc –D DEBUG …
```

Conditional Compilation

- Example: make code more portable

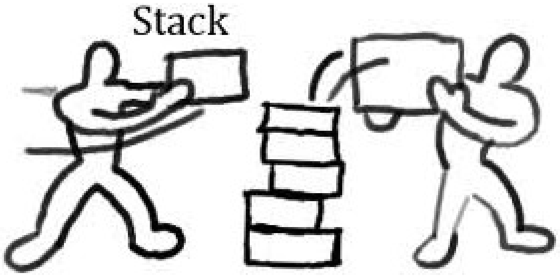
```
#if (OS==WINDOWS) || (OS==windows)
   /* compile this */
#elif (OS==MAC) || (OS==mac)
   /* compile this */
#elif (OS==LINUX) || (OS==linux)
   /* compile this */
#else
   #error “unsupported OS”
#endif
```

```
> gcc –D OS=linux …
```

Conditional Compilation

- Build and run
  - With/without debug info
  - With STACKSIZE 35
  - With stack element type ‘double’

[Toledo: StackV4A]



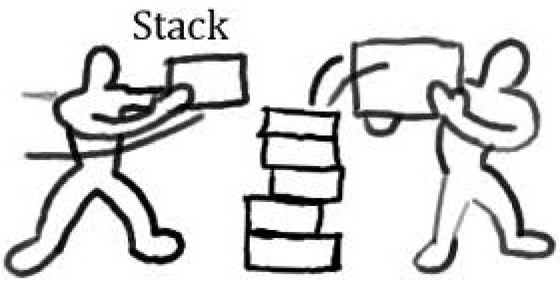
Header Files

- Organize your code in logical ‘modules’ using multiple files
  - Implementation of a module in .c file (source)
  - Interface of a module in .h file (header)
    - Contains all ‘publicly visible’ information of that component
    - Constants, type definitions, function prototypes, …
    - BUT: no implementation code!
  - Other files: #include “component.h”

Header Files

- Study the code ...

[Toledo: Stack_V4B]



[Demo: - assert()

- preprocessor result

- gcc compilation]

Header Files

- Compilation
  - All files at once

    > gcc file1.c file2.c –o prog.exe

  - One-by-one

    > gcc –c file1.c

    – Creates object code for file1: file1.o

    > gcc –c file2.c

    – Creates object code for file2: file2.o

    - Build executable

      > gcc file1.o file2.o –o prog.exe

Header Files

- Header files
  - Not allowed to create circular dependencies: file X includes file Y which includes file X
    - By convention, C programmers surround each header file with one of the following conditionals:

Header guards!

```
#ifndef __MYHEADER_H__
#define __MYHEADER_H__

/*header file content*/

#endif
```

```
#if !defined( __MYHEADER_H__)
#define __MYHEADER_H__

/*header file content*/

#endif
```

Scope and Storage Class

- ‘Scope’ determines where each variable can be used (is ‘visible’) in the program

Scope <> storage class!

```
Int x, y;

Int main ( void )
{
  x = 10;
  y = 10;
  Func();
  ...
}
```

```
Void Func ( void )
{
  Int x;

  x = 1;
  y = 1;
  ...
}
```

Scope and Storage Class

- Local variables
  - Declared within a scope block {…}
  - MUST be declared at the beginning of the scope block!
  - Are only visible and exist within this block

```
{
   Int x, y;
   Char c;

   //statements

   {
      int x;
      // statements
   }
}
```

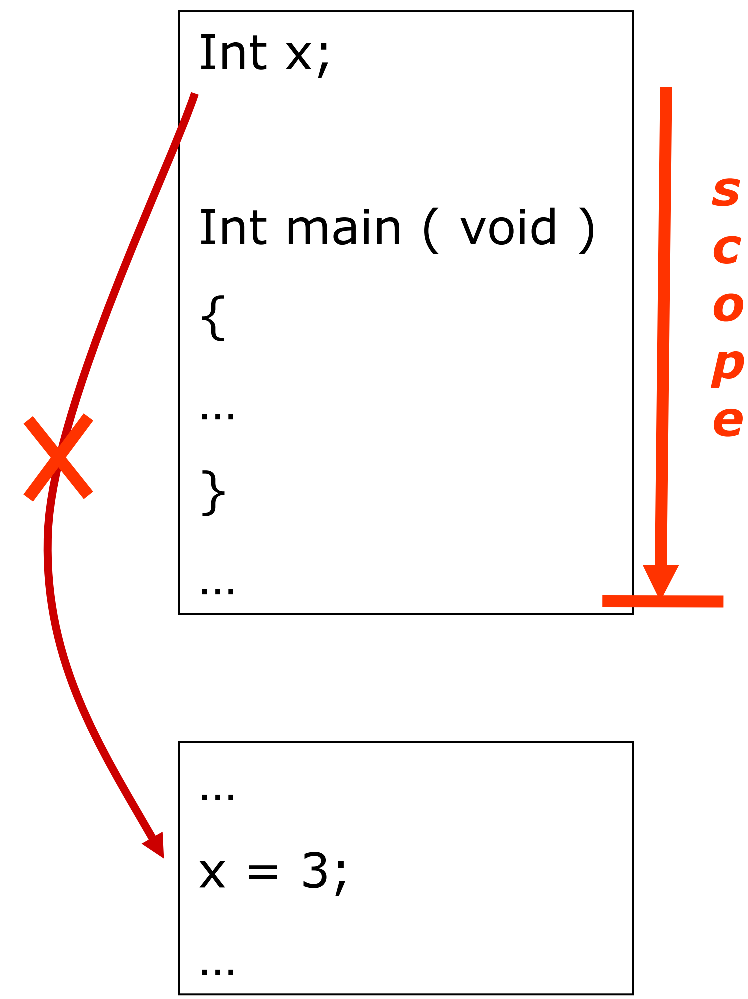
scope

variable declarations

code

Scope and Storage Class

- Global variables
  - Declared outside all scope blocks
    - Sometimes also called ‘external’ variable
    - A global variable is visible from its point of declaration to the end of the file, but using external linkage (‘extern’: see further) can make them visible in other files too



Scope and Storage Class

- Automatic storage class
  - Auto = default storage class of a variable
  - Auto doesn’t change the scope rules
  - Memory allocation and de-allocation (= variable exist) of an automatic variable is done ‘automatically’ by the system
    - Automatic local variables exist only within their scope block
    - Automatic global variables exist during the full execution of the program

Scope and Storage Class

- Extern storage class
  - A source file can reference to a global variable that exists in another source file by declaring that variable using ‘extern’
    - No memory allocation is done (variable must exist already)

Variable declaration <> definition !

  - Function can be ‘extern’ too: cf. global variables
    - Function prototypes are by default ‘extern’

Scope and Storage Class

- Static storage class
  - Static global variables exist during the execution of the program but are only visible within their own source file
    - External linkage of these variables is prevented
  - Static local variables exist the whole time the program is executing
    - The scope is the same as automatic variables
  - Static variables are initialized only the first time (by default it is initialized to 0)

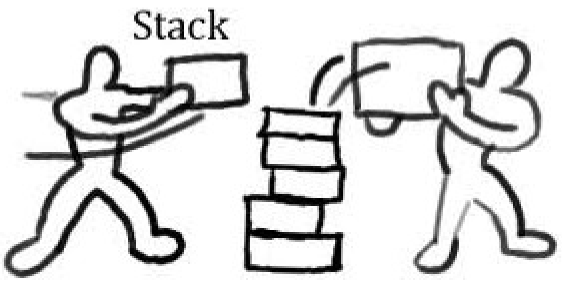
Scope and Storage Class

- Static storage class
  - A function can be ‘static’ which prevents external linkage of the function (cf. global var)

Scope and Storage Class

- Study the code …

[Toledo: Stack_V4C]



Macros

No spaces!

```
#define identifier(param-list) (replacement-text)
```

- Example
  - Macro: `#define min(x,y) x<y?x:y`
  - Function:

```
int min_func(int a, int b)
{
   return ( (a < b)? a : b );
}
```

Macros

- BUT: be aware that a macro uses text substitution!
  - Example

```
2 * min(a,b)    ==>>    2*a < b ? a : b
```

  - Better

```
#define min(x,y) ((x)<(y)?(x):(y))
```

Macros

- # in macro
  - Makes a string of a macro-parameter
  - Example:

```
#define PRINTSUM(x,y)
        printf(#x " + " #y " = %d\n", x+y);

        PRINTSUM(1+2,4);

           ==>> 1+2 + 4 = 7
```

Macros

- Macro with multiple statements
  - Example: after a free(), the pointer should be set to NULL
    - C statements

```
free( ptr );
ptr = NULL;  <== often forgotten!
```

    - Define a macro: version 1

```
#define FREE(p) free(p);p=NULL;
```

But not always correct ...

```
if ( ... )
    FREE(ptr);
```

```
If (…)
    free(p);p=NULL;
```

<u>Which is the same as:</u>

```
If(...)
    free(p);
p=NULL;
```

Macros

- Macro with multiple statements
  - Define a macro: version 2

```
#define FREE(p) { free(p); p=NULL; }
```

But not always correct ...

```
if ( ... )
    FREE(some_ptr);
else
    do_something_else;
```

... will fail on 'else'

```
If (…)
    {free(p);p=NULL;};
Else
    do_something_else;
```

Macros

- Macro with multiple statements
  - Define a macro: version 3 – classical trick

```
#define FREE(p) do { free(p);p=NULL;} while(0)
```

  - – Do-while makes one statement of it!

Macros

- Macro with multiple statements
  - A debug macro

```
#define DEBUG_PRINT(...)                                          \
    Do {                                                          \
        printf("In %s in function %s at line %d: ",               \
                        __FILE__, __func__, __LINE__);            \
        printf(__VA_ARGS__);                                      \
    }   while(0)
```

In code: DEBUG_PRINT(“error: stack is full (maxsize = %d)”, MAXSIZE);

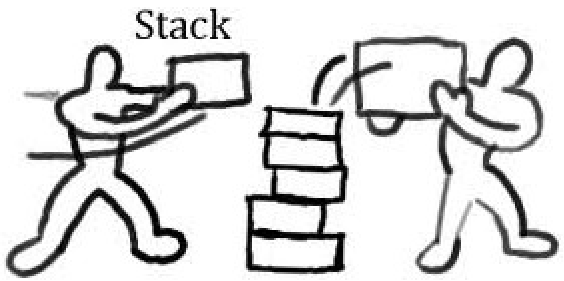
Macros

- #pragma
  - Used to define directives for the compiler
    - Always implementation/compiler-dependent
  - Pragma directive is ignored if the compiler doesn’t recognize it
  - Example
    - #pragma GCC poison printf

Macros

- Study the code …

[Toledo: Stack_V4D]



Macros

- Why using macros?
  - Macro is kind of 'type-less function'
  - Eliminate function call overhead
    - Counter-argument: use 'inline' function (since C99)
      - E.g.: inline my_func(...);
      - But: inline is only a 'request' to the compiler to inline the function ...
  - Token passing

```
#define STACK_DEF(stack_type)                   \
        typedef struct stack_type##_stack {     \
                stack_type data[SIZE];          \
                int top;                        \
        } stack_type##_stack_t

STACK_DEF(int);
STACK_DEF(double);
```

  - Use compile-time info at run-time
    - E.g. __FILE__, __LINE__, __func__, ...

LECTURE C4

Low-Level Operations

Representation

- Constant and literals
  - Examples
    - Unsigned: printf("%u\n", 13u);
    - Long: printf("%ld\n", 12345L);
    - Long long: printf("%lld\n", 123456789LL);
    - Long double: printf("%Lg\n", 11.2L);
    - Octal: printf("\\o%o\n", 077);
    - Hex.: printf("0x%x\n", 0x7B2F);
- C has no data type 'byte'
  - Typedef unsigned char byte_t;
    - Assuming sizeof(char) = 1

Representation

- Size of int, float, double, etc. is *system-dependent*
  - = differences in byte size, endianness, bit representation, ...
    - Check <limits.h>, <float.h>, etc.
    - Use sizeof(...) operator
      - E.g. sizeof(int) : return #bytes used for int on this system

Portable Integer Types

- C99 standard defines <inttypes.h>
  - Contains also macros for printf/scanf of these new types

int8_t	8-bit signed integer
uint8_t	8-bit unsigned integer
int16_t	16-bit signed integer
uint16_t	16-bit unsigned integer
int32_t	32-bit signed integer
uint32_t	32-bit unsigned integer
int64_t	64-bit signed integer
uint64_t	64-bit unsigned integer

intptr_t	signed integer which can hold the value of a pointer
uintptr_t	unsigned integer which can hold the value of a pointer

INT8_MAX, INT16_MIN, UINT32_MAX, etc. define constants holding min/max values

Bit Operators

- & : and
- | : or
- ~ : one's complement
- ^ : xor
- << : left-shift
  - x << 1 = 2*x
- >> : right-shift
  - Unsigned: 0's are inserted
  - Signed: 0's or 1's might be inserted (machine dependent)
  - The result of a left/right shift of a negative number or of data size or more bits is undefined in C

Arguments are integers: short, long, long long, signed, unsigned

>> machine dependent!

Check Bit

```
#define IRQ_FLAG 0x10 /* bit mask */

typedef unsigned char Byte;

Byte byte;

if (byte & IRQ_FLAG) {
     HandleIRQ();
}
```

Byte:	10111001
Bit mask:	00010000
&:	00010000

Works also for a group of bits!

Select Bit

```
#define MASK 0x10

d = (byte & MASK) >> 4;
```

byte:	10111001
Bit mask:	00010000
& + >>:	00000001

Set Bit

```
#define MASK 0x10

byte |= MASK; /* Set bit */
```

Works also for a group of bits!

byte:	10101001
Bit mask:	00010000
|:	10111001

Reset Bit

```
#define MASK 0x10

byte &= ~MASK;    /* Clear bit */
```

Works also for a group of bits!

byte:	10111001
Bit mask:	00010000
~ + &:	10101001

XOR Swap

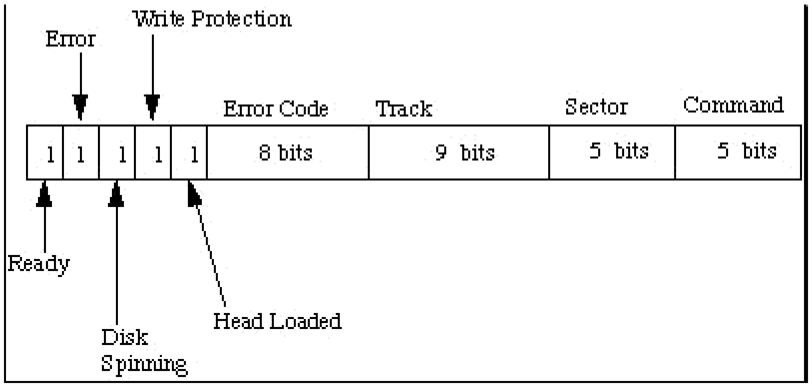
```
int x,y,temp;
temp = x;
x = y;
y = temp;
```

```
x ^= y;
y ^= x;
x ^= y;
```

Bit Fields in Structs

- Only for unsigned and int
  - Machine-dependent behaviour!

```
struct DISK_REGISTER  {
     unsigned ready:1;
     unsigned error_occured:1;
     unsigned disk_spinning:1;
     unsigned write_protect:1;
     unsigned head_loaded:1;
     unsigned error_code:8;
     unsigned track:9;
     unsigned sector:5;
     unsigned command:5;
};

struct DISK_REGISTER reg;
...
if (reg.ready) { … }
```



Unions

```
Union int_or_float {
    int i;
    float f;
  } n;
```

```
n.i = 4444;
Printf("i = %d – f = %e\n", n.i, n.f
);
```

➔ i = 4444 – f = 0.622737e-41

```
n.f = 4444.0;
Printf("i = %d – f = %e\n", n.i, n.f
);
```

➔ i = 1166729216 – f = 4.444e+3

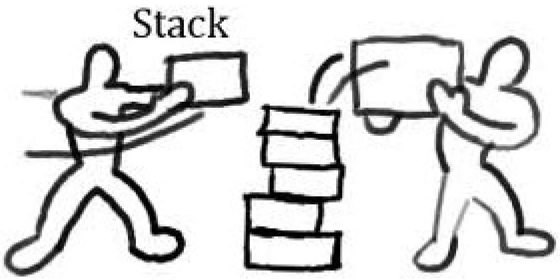
Bit fields can also be used in unions

Unions

- Example: bytes and bits

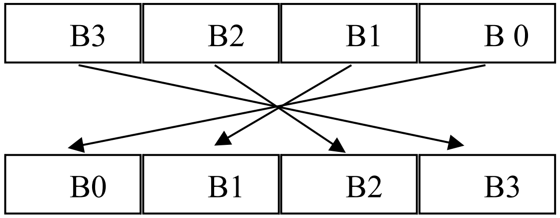
```
struct bits {
    unsigned char b1:1, b2:1, b3:1, b4:1, b5:1, b6:1, b7:1, b8:1;
};

union myByte {
    unsigned char byte;
    struct bits bit;
};

union myByte by;

by.byte = 0xFF;
by.bit.b1 = 0;
by.bit.b2 = 0;
```

[DEMO: StructUnion]



Unions

- Example: little/big endian
  - unsigned = B0 B1 B2 B3
  - Little endian:
  - Big endian:




```
union {
    unsigned value; //assuming 32bit unsigned
    unsigned char byte[4];
} x;
x.value = 0x11223344;
printf( "b0=0x%x \tb1=0x%x \tb2=0x%x \tb3=0x%x \n",
        x.byte[0], x.byte[1], x.byte[2], x.byte[3] );
```

Storage Class: Register

- Register
  - Example: register int x;
  - Request to store the variable in a register to optimize speed
    - It is not guaranteed that the variable is indeed mapped on a register – often ignored by compiler
  - Register variables have the same scope and existence properties as automatic variables

Qualifier: Volatile

- Qualifier in variable declarations
  - Example: volatile int x;
- Indicates to the compiler that a variable might change due to some “external” action, e.g.:
  - Variable is changed by an ISR
  - Variable is shared with other threads
  - Variable is memory-mapped to a peripheral register of some device

>> prevents compiler to apply optimizations on this variable

Qualifier: Volatile

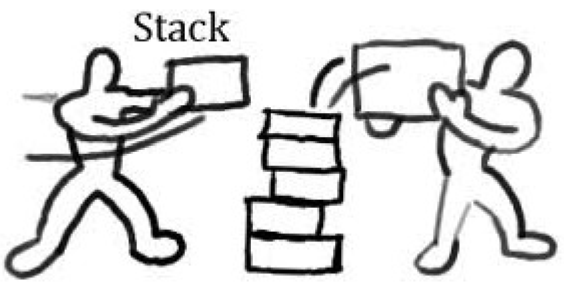
- Example

```
#define REGADDR 0xFF670EB2

typedef unsigned char Byte;

Byte volatile *p = REGADDR;
//or: volatile Byte *p = REGADDR

while ( *p == 0 ) {
   // busy loop: wait until data in register
}

// now do something with *p
```

Qualifier: Volatile

- Study the code …

[DEMO: Volatile]



Calling Assembly

- Gcc, x86: www.ibm.com/developerworks/library/l-ia.html

```
int main() {
  int arg1, arg2, add ;

  printf( "Enter two integer numbers : " );
  scanf( "%d%d", &arg1, &arg2 );

  /* Perform Addition */
  asm("addl %%ebx, %%eax;":"=a" (add) :"a" (arg1), "b" (arg2));

  printf( "%d + %d = %d\n", arg1, arg2, add );
  return 0 ;
}
```

Calling Assembly

- Gcc, x86:
  - Assembly template

```
asm ( “statement” … “statement”
      : output operands (optional)
      : input operands (optional)
      : list of clobbered registers (optional)
     );
```

Calling Assembly

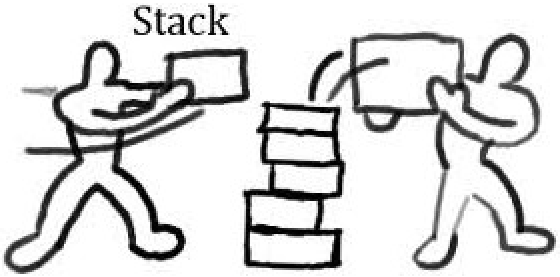
- Gcc, x86:
  - Volatile: no (compiler) optimizations allowed

```
asm ( "assembly code“ );
__asm__ ( "assembly code“ );
__asm__ volatile ( "assembly code“ );
__asm__ __volatile___ ( "assembly code“ );
```

Calling Assembly

- Compiling C code to assembly:
  - Gcc -S file.c
- Compiling C and assembly code:
  - Gcc file.c assembly.s

[DEMO]



LECTURE C5

Dynamic Memory and Structures

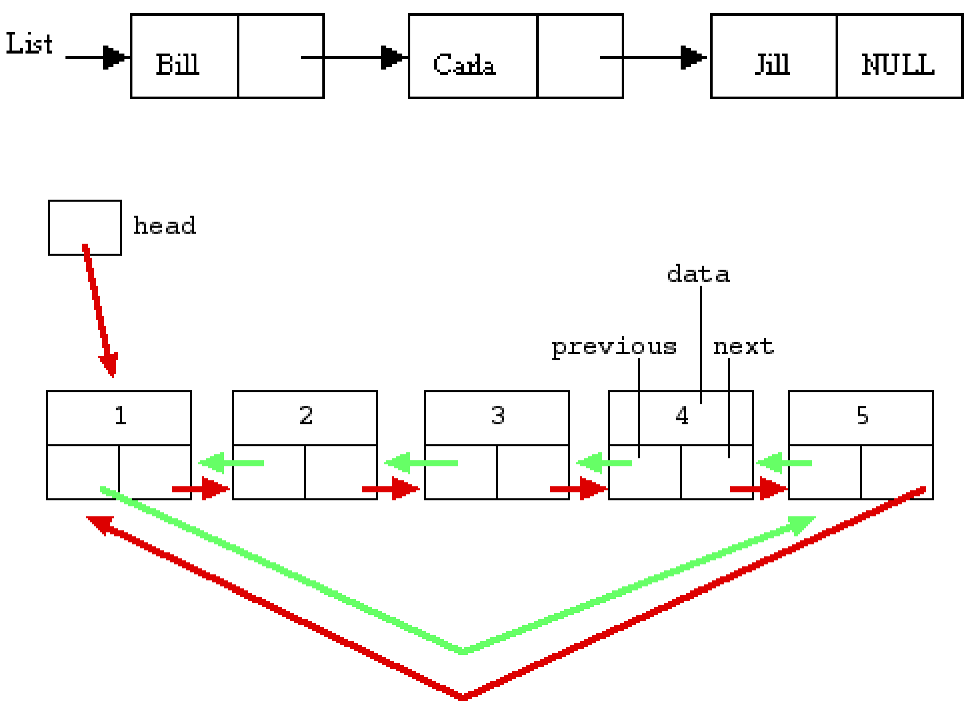
Dynamic structures

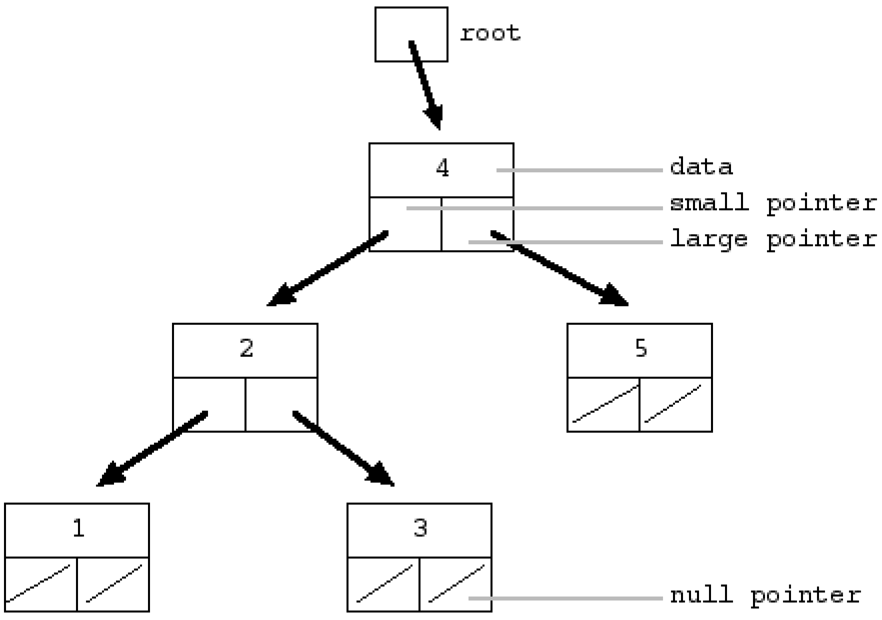
- Example

```
typedef struct {
  int x;
  int y;
  float f;
} my_struct_t;

my_struct_t *p;
p = (my_struct_t *)malloc( sizeof(my_struct_t) );
if ( p != NULL ) {
  p->x = 1;          // not: (*p).x = 1;
  p->y = 2;
  p->float = 0.5;
}
```

Dynamic structures

- Easy to create complicated dynamic data structures using pointers to structs
  - Linked lists, binary trees, graphs, …





Dynamic structures

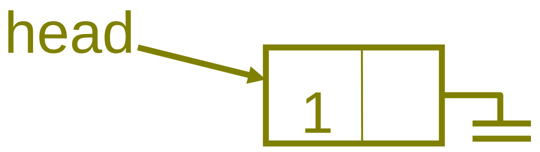
- Example: single linked list structure

```
typedef struct node {
   int data;
   struct node * next;
} node_t;

typedef node_t *  list_t;

list_t head;

// an empty list
head = NULL;
```

Dynamic structures

- Example: single linked list structure

```
// add the first element
head = (list_t) malloc( sizeof( node_t ) );
assert( head );

head->data = 1;
head->next = NULL;
```



Dynamic structures

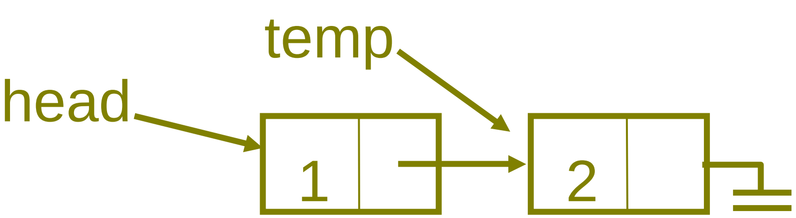
- Example: single linked list structure

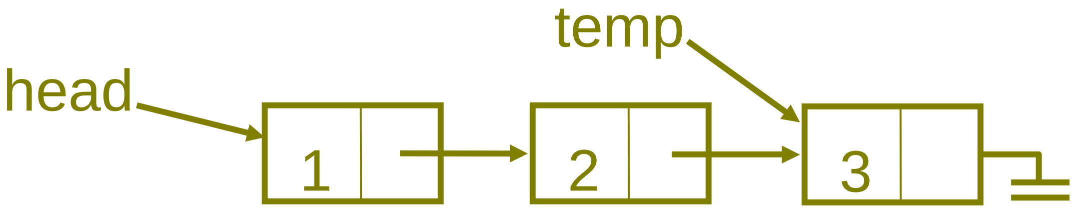
```
// add a second element
head->next = (list_t) malloc(sizeof(node_t));
assert( head->next );

head->next->data = 2;
head->next->next = NULL;
```

Dynamic structures

- Example: single linked list structure

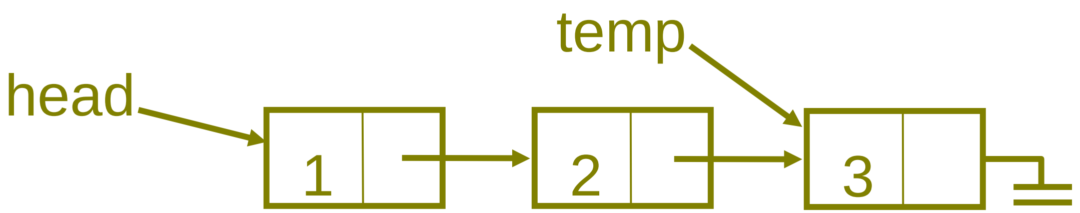
```
// add a third element
list_t temp;
temp = head->next;
temp->next = (list_t) malloc(sizeof(node_t));
assert( temp->next );

temp->next->data = 3;
temp->next->next = NULL;

// to insert the next element
temp = temp->next;
```





Dynamic structures

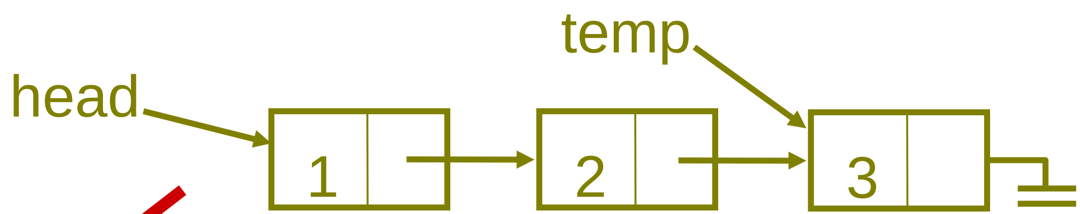
- Example: single linked list structure




```
// delete the third element
free(temp);

// BUT end-of-list needs to be indicated by NULL!
// This is not possible anymore!
```

Dynamic structures

- Example: single linked list structure



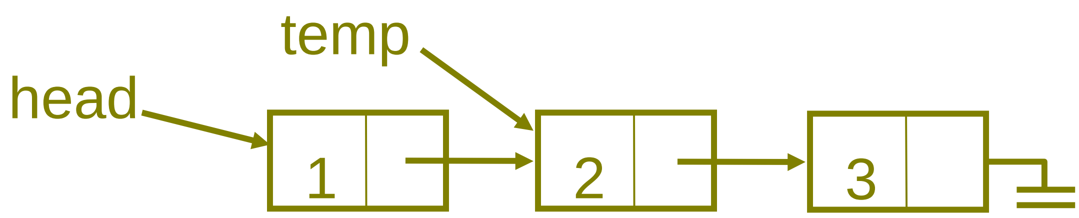

```
// delete                  element
free( te
```

WRONG!

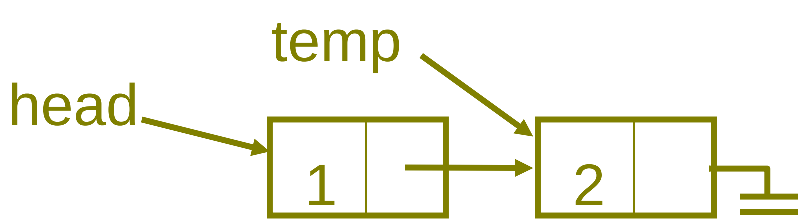
```
// BUT end-of-list needs to be indicated by NULL!
// This is not possible anymore!
```

Dynamic structures

- Example: single linked list structure



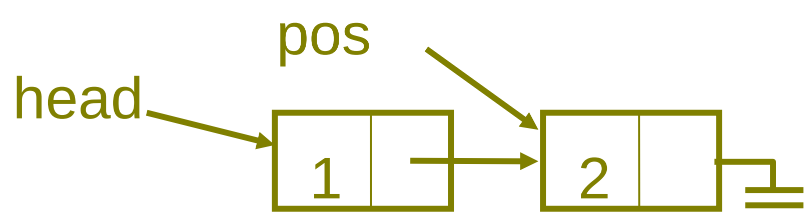

```
// delete the third element
temp = head->next

free( temp->next );
temp->next = NULL;
```



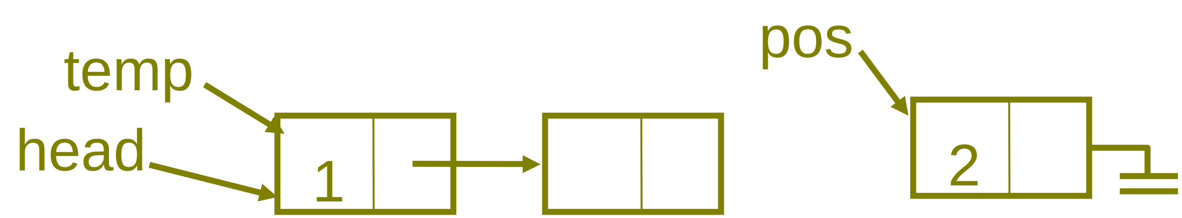
Dynamic structures

- Example: single linked list structure

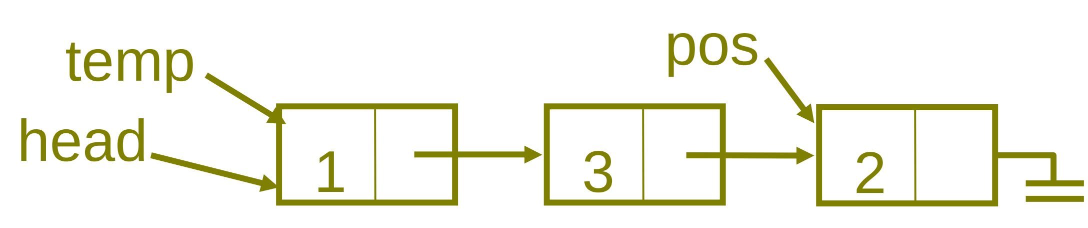
```
// inserting an element at ‘pos’
```




```
temp = head;
temp->next = (list_t) malloc(sizeof(node_t));
assert( temp->next );
```



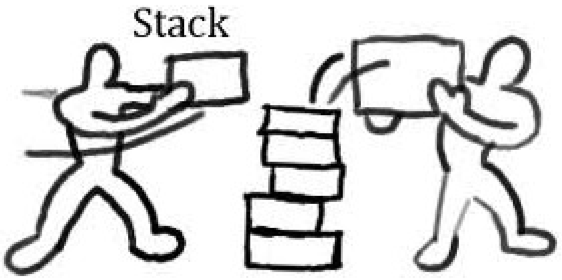

```
temp->next->data = 3;
temp->next->next = pos;
```



Dynamic structures

- Study the code ...

[Toledo: StackV6]



Dynamic structures

- More examples, memory drawings and solutions can be found in the following papers (Toledo)
  - “Linked List Basics”, Nick Parlante, Standford University
  - “Linked List Problems”, Nick Parlante, Standford University

More Memory Functions

- More memory-related functions
  - void *calloc(size_t nmemb, size_t size);
    - Allocate memory for an array of 'nmemb' elements of size 'size' and initializes memory to zero
  - void *memcpy(void *dest, const void *src, size_t n);
  - void *memmove(void *dest, const void *src, size_t n);
  - int memcmp(const void *s1, const void *s2, size_t n);
  - void *memchr(const void *s, int c, size_t n);
    - Search c in first n bytes of memory s
  - void *memset(void *s, int c, size_t n);
    - Set the first n bytes of memory s to byte c

Check C manual reference for more possibilities!

Common Memory Errors

- Common errors
  - Pointer points to no or not enough allocated memory
  - Example

```
int *ptr;
…
*ptr = x;
```

```
char str[] = “luc”;
…
strcpy(str,“peter”);

==> strncpy, strdup, ...
```

Common Memory Errors

- Common errors: dangling pointers
  - Pointing to memory that no longer exists
  - Using freed memory

```
char *ptr, *str;
str = malloc(…);
strcpy(str, “luc”);
…
ptr = str;
…
free(ptr);
…
printf(“%s\n”, str);
```

```
int *p;  // global var

Void func(void){
  int a;
  p = &a;
}
…
func();
printf(“%d\n”, *p);
```

Common Memory Errors

- Common errors
  - Memory leaks
    - Neglecting to free disused blocks
    - Eventually the system will run out-of-memory

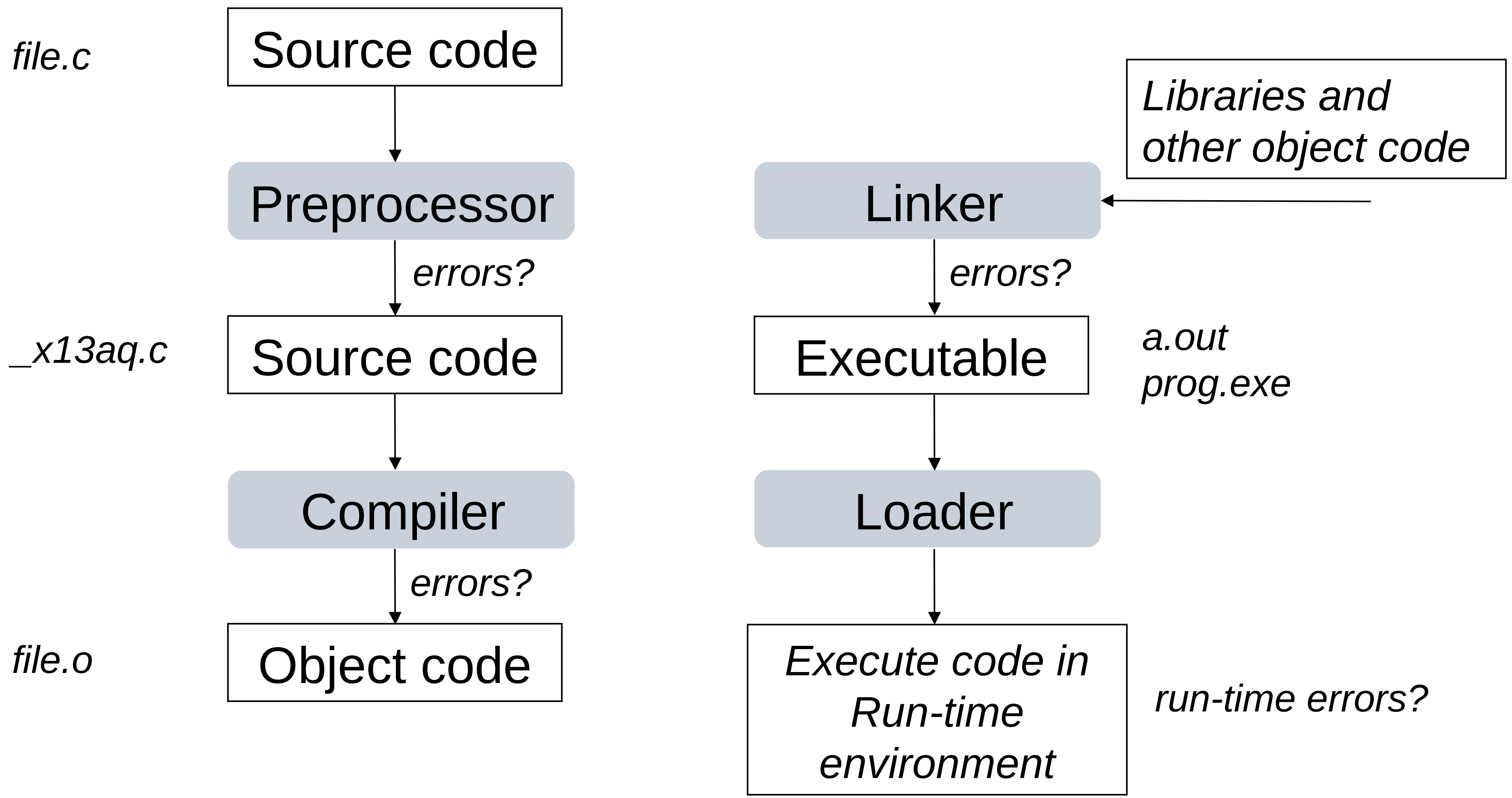
=> Use memory debugging tools such as Valgrind

  - (see lab sessions)

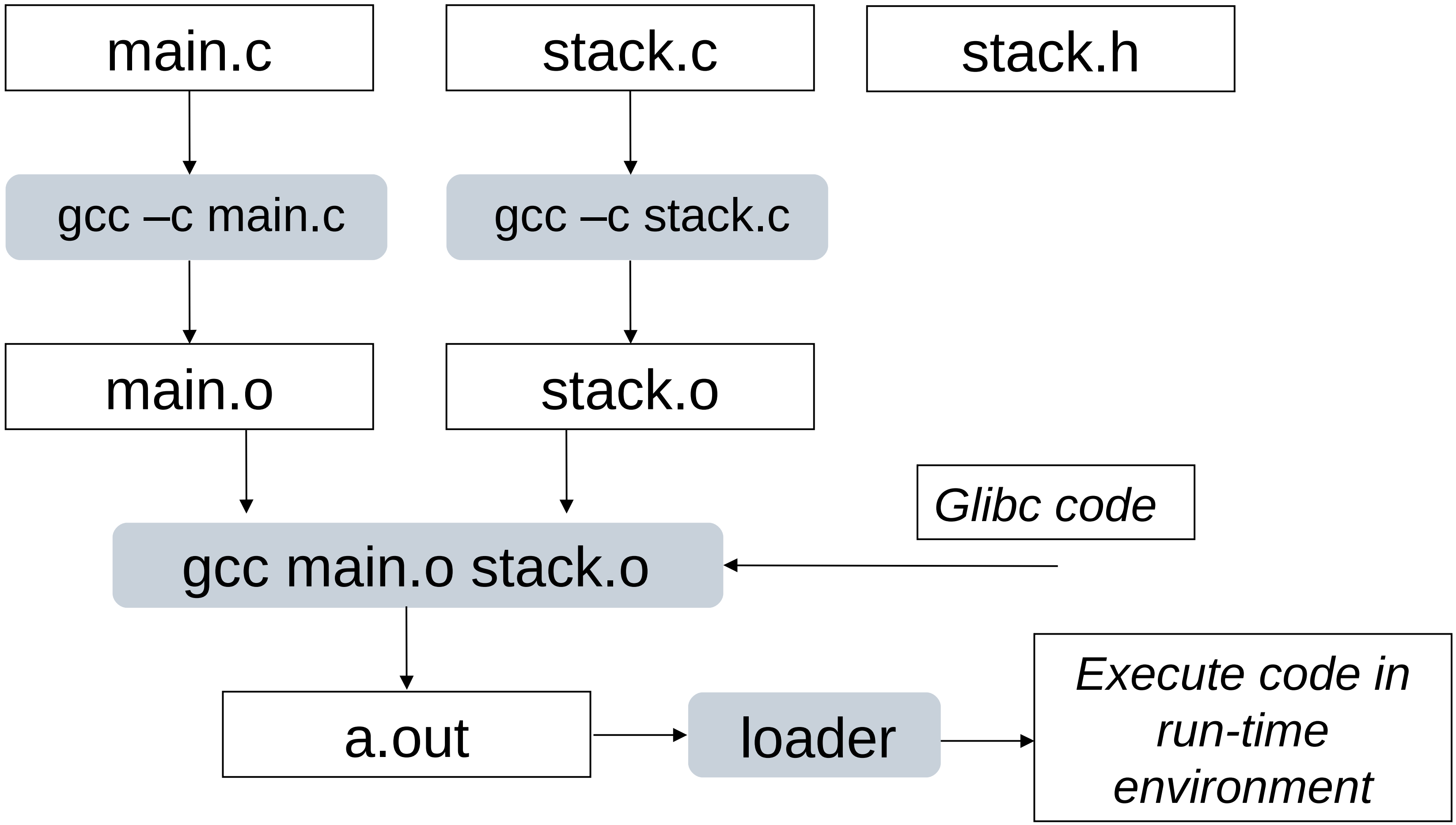
LECTURE C6

When Programs Become Larger

The Build Process



The Build Process

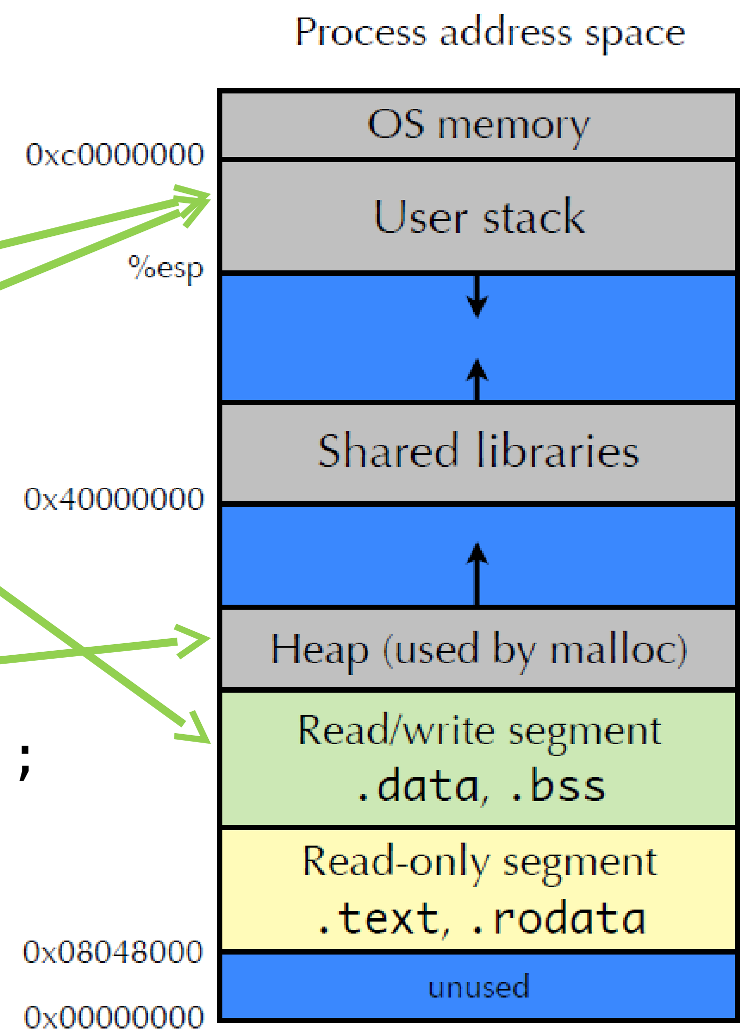


The Build Process

- Calling the preprocessor
  - Gcc –E file.c
  - Cpp file.c
- Calling the compiler
  - Gcc –c file.c
- Calling the assembler
  - Gcc –S file.c
    - Gcc calls 'as' utility
- Calling the linker
  - Gcc file1.o file2.o –lsomelib
    - Gcc calls 'ld' utility

Memory Layout

```
int global_static;

void foo(int auto_param)
{

   int auto_i, auto_a[10];

   double *auto_d =
          malloc(sizeof(double)*5);
}
```

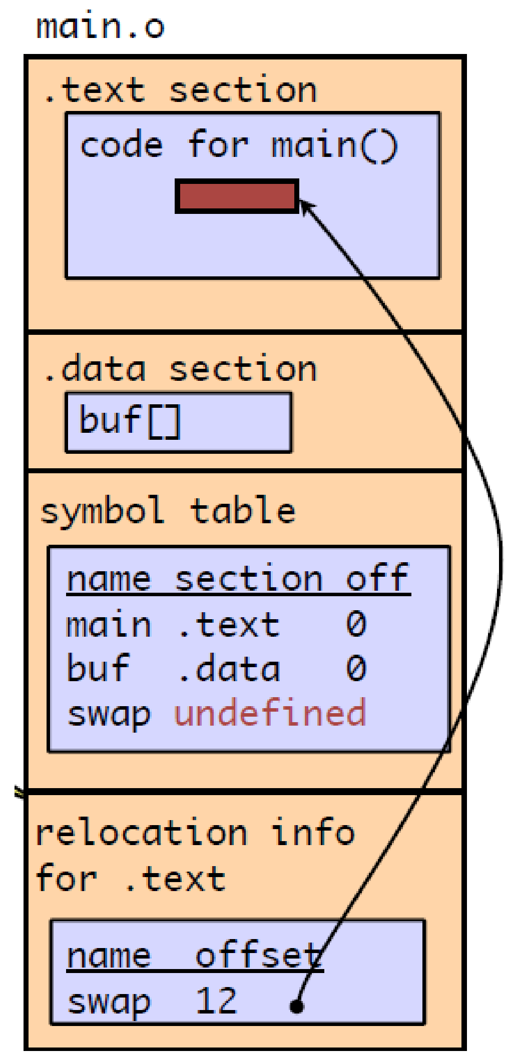


Memory layout

- In Linux: /proc/<process_id>/maps
  - Use ‘ps aux’ to see processes & their ids
  - Do ‘cat /proc/<process_id>/maps’
  - Overview of memory layout of the process <process ID>
  - See ‘man proc’ for details

Compiler

- Compiler
  - Translate source code into object code
    - Syntax checking
  - Symbol table for global variables and function names
  - Several object code formats exist
    - ELF, COFF, PE COFF, ...
  - Example:



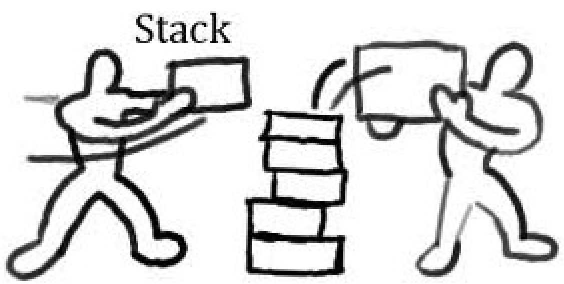
main.c

```c
int buf[2] = {1,2};

int main()
{
  swap();
  return 0;
}
```

Compiler

- Compiler
  - Symbols are mapped to memory regions
    - Text, read-only memory, BSS, uninitialized, ...
    - They have no addresses yet (address 0 or an offset value)
  - Undefined symbols are 'defined' externally (other object code file or library)
- Nm-tool
  - List symbols from object files
  - Gcc -c main.c
  - Nm main.o

[DEMO]

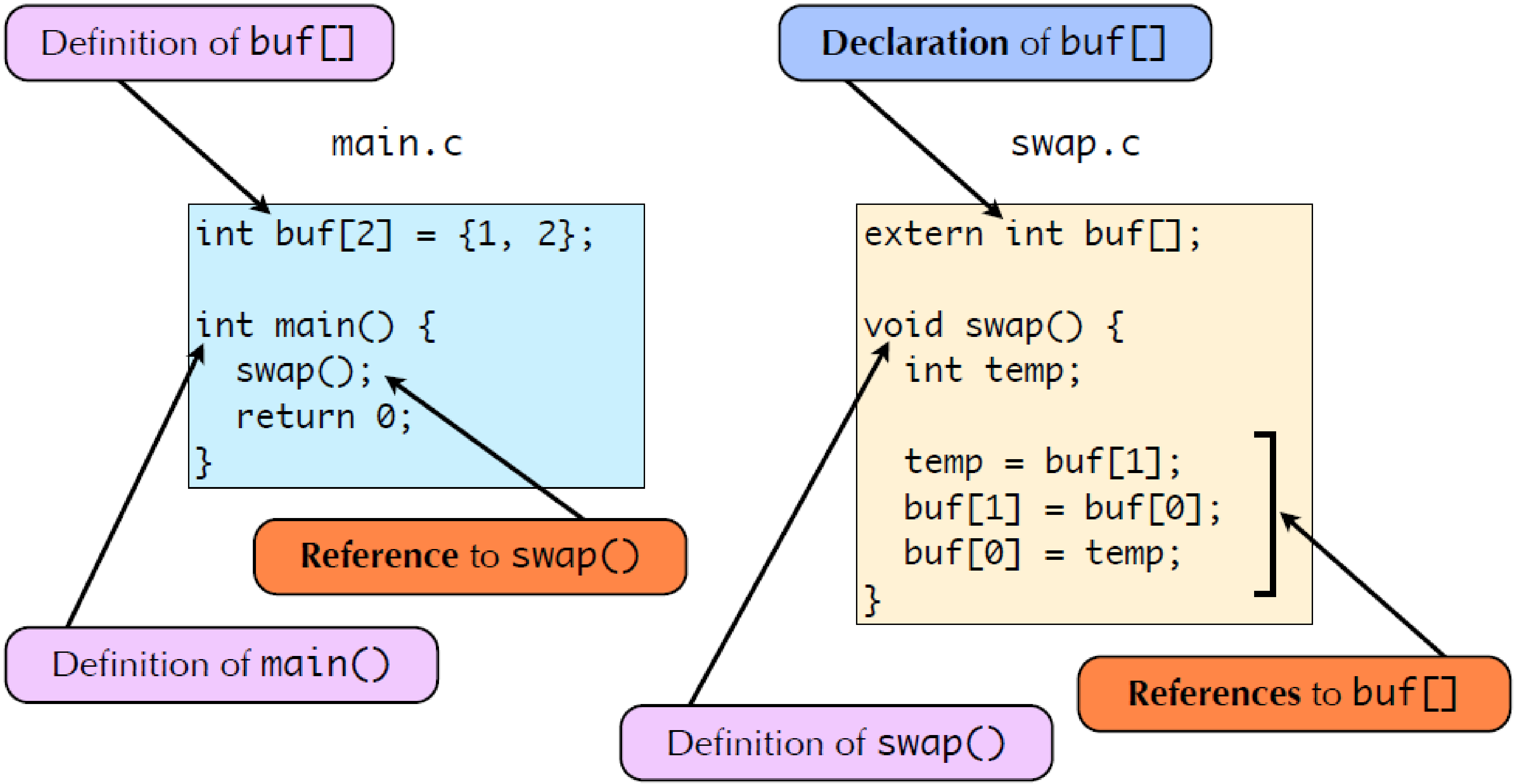


Linker

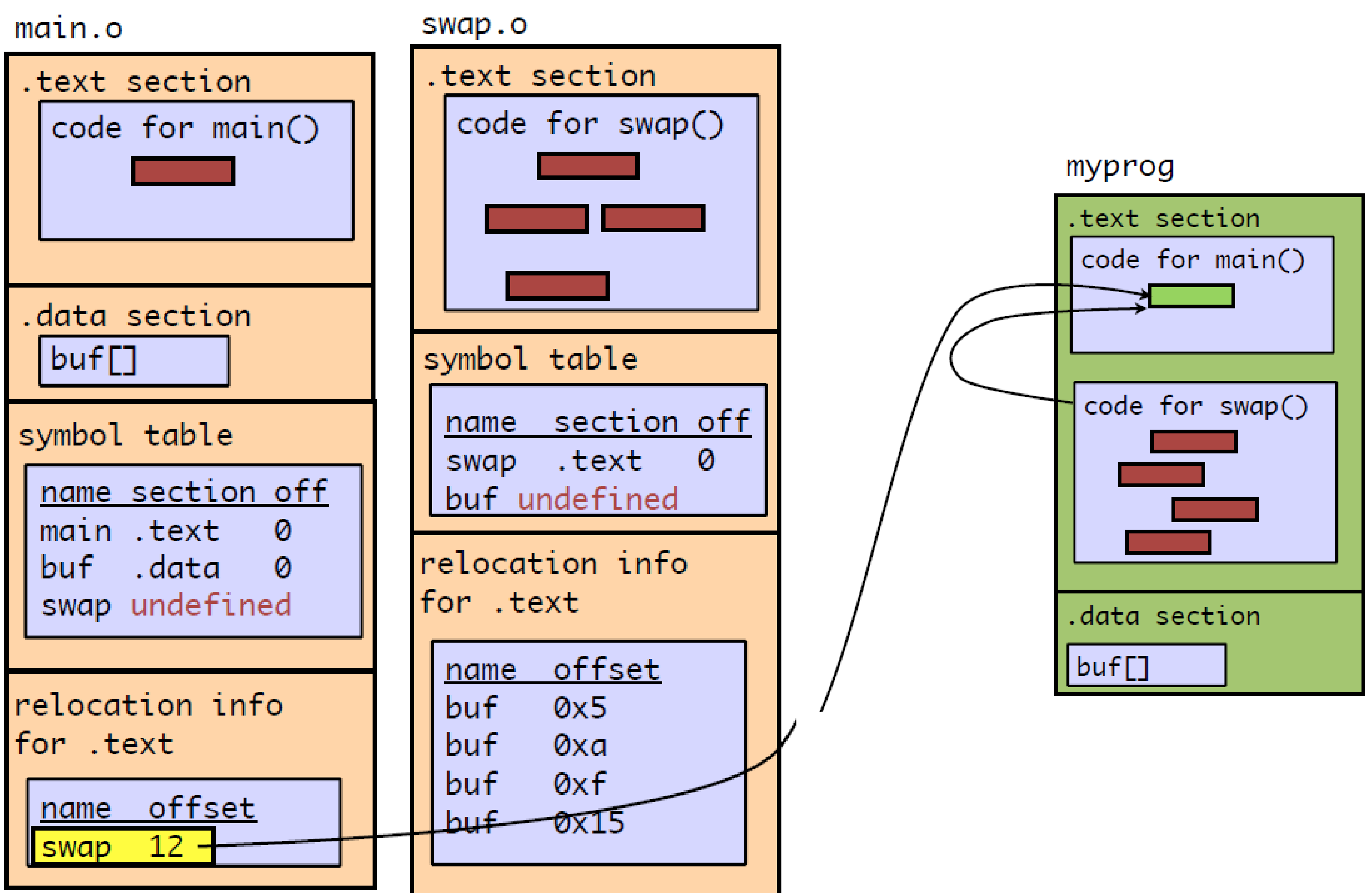
- Combine multiple object files into an executable that can be loaded in memory and executed

- Linker tasks
  - **Copy** code and data from each object file to the executable
  - **Resolve** references between object files
    - Undefined symbols
  - **Memory allocation** to defined static symbols
    - Symbols from shared libraries get memory at run-time

Linker


Definition of buf[]
main.c
int buf[2] = {1, 2};
int main() {
swap();
return 0;
}
Reference to swap()
Definition of main()
Declaration of buf[]
swap.c
extern int buf[];
void swap() {
int temp;
temp = buf[1];
buf[1] = buf[0];
buf[0] = temp;
}
References to buf[]
Definition of swap()

Linker

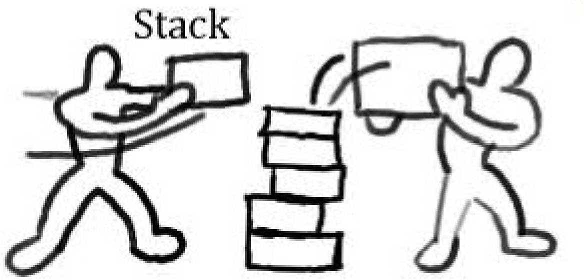


main.o
.text section
code for main()
.data section
buf[]
symbol table
name section off
main .text 0
buf .data 0
swap undefined
relocation info
for .text
name offset
swap 12
swap.o
.text section
code for swap()
symbol table
name section off
swap .text 0
buf undefined
relocation info
for .text
name offset
buf 0x5
buf 0xa
buf 0xf
buf 0x15
myprog
.text section
code for main()
code for swap()
.data section
buf[]

Linker

- Nm-tool
  - Link stack.o and main.o to a.out
  - Nm a.out

[DEMO]



Linker

- Typical linker errors
  - Duplicate reference to variables or functions
  - Undefined symbols

- Static, extern, register, … influence the linkage of variables and functions

- Volatile, const, ... used by the compiler

Linker

- Why a compilation and linking stage?
  - Time efficiency
    - Faster building large programs
      - Only compile files that were changed
    - Use precompiled code (library code)
  - Memory efficiency
    - Allows the use of shared code (libraries)

Loader

- OS starts loading process when program wants to run
  - A new process is created (see part 3 – execve() syscall!)
  - Process execution environment is created
    - Memory is allocated
      - On modern OS this is virtual memory
    - Data and code is copied to memory
      - Locate and load shared lib code
    - The function stack is initialized
    - The processor’s program counter (PC) is set to the executable's entry point (calls main())

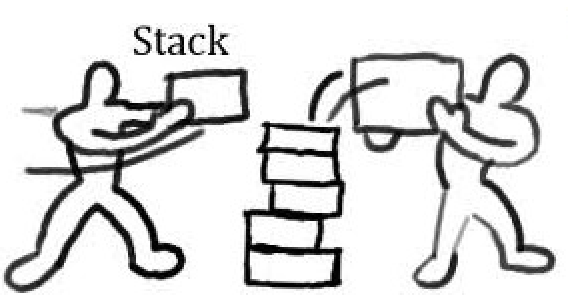
The Build Process

- Objdump-tool
  - Disassembly of object code
    - Objdump –d swap.o
  - Disassembly of executable
    - Objdump –t a.out

>> see man objdump for more options

```
$ objdump -d swap.o
...
00000000 <swap>:
0:   55                              push   %ebp
   1:    89 e5                          mov    %esp,%ebp
   3:    8b 15 04 00 00 00              mov    0x4,%edx
   9:    a1 00 00 00 00                 mov    0x0,%eax
   e:    a3 04 00 00 00                 mov    %eax,0x4
  13:    89 15 00 00 00 00              mov    %edx,0x0
  19:    5d                             pop    %ebp
  1a:    c3                             ret
```

[DEMO]



Libraries

- Target: allow code reusage and code sharing
  - Solution 1: make source code available
    - But: everybody needs to recompile and relink
    - But: must give source code
  - Solution 2: make .o files available
    - But: everybody needs to relink
    - But: not efficient when code consists of many .o files

Libraries

- Solution 3: package code together in library
  - Static library
    - .a files in Linux
  - Shared library
    - .so files in Linux
    - .dll files in Windows

Libraries

- Static library
  - Archive .o files in .a library
  - Examples
    - libc.a (the C standard library)
      - archive of mare than 1500 object files.
      - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math
    - libm.a (the C math library)
      - archive of more than 400 object files.
      - floating point math (sin, cos, tan, log, exp, sqrt, …)

Libraries

- Static library
  - How? Use ar-tool
    - Example: ar rs libtest.a file1.o file2.o file3.o
    - What's in lib?
      - E.g. ar t /usr/lib/x86_64-linux-gnu/libc.a
  - Using static libraries
    - Example
      - gcc main.c file.c libtest.a
      - Gcc main.c file.c -ltest
      - Gcc main.c file.c -ltest -L/path_to_lib

Libraries

- Shared library
  - Drawbacks of static libraries
    - Code duplication
      - E.g. all C programs with ‘printf’ have own printf-code in the executable
        - Big executable files
    - Updates of library require relinking of programs using the library

Libraries

- Shared library
  - Shared library needs to be located at compile-link time
    - Lookup symbols in external libraries
    - Type checking, etc.
  - Shared library needs to be located again at run-time for loading
    - Done by run-time linker in ‘ld’ lib
    - Load symbols from shared library are allocated memory

Libraries

- Shared library
  - How to create a shared lib
    - Gcc -fPIC -c file1.c file2.c
      - PIC = position independent code
    - Gcc –shared –o libtest.so file1.o file2.o

Libraries

- Shared library
  - How to link program with shared lib?
    - gcc main.c /path_to_lib/libtest.so
      - Location of lib at compile-link time!
    - Better:
      - Gcc main.c -L/path_to_lib/ -ltest

Libraries

- Shared library
  - How to run program with shared lib?
  - Location of lib at run-time loading!
    - Check shared lib dependencies: ldd prog.exe
    - Option 1 (ok for testing/debugging): set the run path
      - Gcc main.c -L/path_to_lib/ -Wl,-rpath=/path_to_lib/ -ltest
    - Option 2 (better, more permanent): copy lib to default lib-folder (admin perm.)
      - E.g.: cp libtest.so /usr/lib
        - Check permissions of libtest.so
      - Update the cache of the loader
        - Run 'ldconfig'
      - Gcc main.c -ltest

Libraries

- Dynamic loading: load shared code **on demand** at runtime
  - Advantages
    - Extend program functionality after compilation and linking
  - Example: browser plugins

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

int main() {
    void *handle;
    void (*somefunc)(int, int);
    char *error;

    /* dynamically load the shared
     * lib that contains somefunc() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
     fprintf(stderr, "%s\n", dlerror());
     exit(1);
    }
    /* get a pointer to the somefunc()
     * function we just loaded */
    somefunc = dlsym(handle, "somefunc");
    if ((error = dlerror()) != NULL) {
     fprintf(stderr, "%s\n", error);
     exit(1);
    }

    /* Now we can call somefunc() just
     * like any other function */
    somefunc(42, 38);

    /* unload the shared library */
    if (dlclose(handle) < 0) {
     fprintf(stderr, "%s\n", dlerror());
     exit(1);
    }
    return 0;
}
```

Libraries

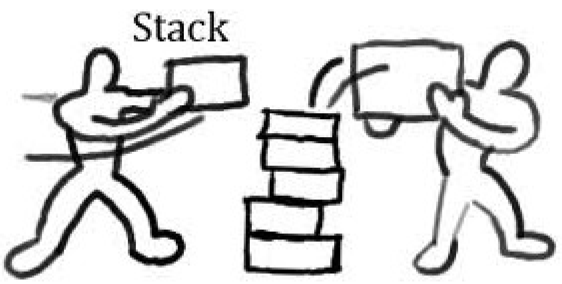
- Example: dynamic link loader
  - Dynamic link loader can be called to do this
    - 4 functions implement the interface to the loader
      - Dlopen() : load dynamic library in memory
      - Dlclose() : unload the lib
      - Dlsym() : look up and return addresses to symbols (e.g. functions) in lib
      - Dlerror() : used for error handling
    - Build code with: gcc ... -ldl

Libraries

Study the code …

- Build: gcc main.c -ldl
-
- Check with ldd: no dependency on libstack.so!

[Toledo: Dlopen]



Library linking: summary

- Linking can be done at
  - Compile time (static linking)
  - Run time (dynamic linking)
    - 'a.out' is incomplete executable
  - On demand at run time (dynamic loading)
- Gcc uses dynamic linking of libraries by default
  - Static linking can be enforced using '-static' flag, e.g. 'gcc -static ...'

Make Files

- Compilation of large code trees can take a lot of time (e.g. Linux kernel)
  - Make utility eases the build process
  - Make avoids compiling files that didn't change

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

main.o: main.c stack.h
	gcc -Wall -c main.c

stack.o: stack.c stack.h
	gcc -Wall -c stack.c

clean:
	rm myprog *.o
```

Tab!

Make rule:

target: dependencies

action(s)

Idea: check for updates of the dependencies ***and*** *rebuild myprog when the modification timestamp of any of the dependencies is more recent than the modification timestamp of myprog.*

Make Files

- How to run a make file?
  - Type 'make'
    - Make will look for a file 'makefile' and will run the commands of the first rule/target
    - If there is no file 'makefile', it will look for the file 'Makefile' and will run the commands of the first rule/target
  - Type 'make -f <someFile>'
    - Make will run the commands of the first rule/target in 'someFile'
  - Type 'make <targetName>'
    - Make will run the commands of the rule/target <targetName> in the file makefile (or Makefile)
    - Example: make -f mymake clean

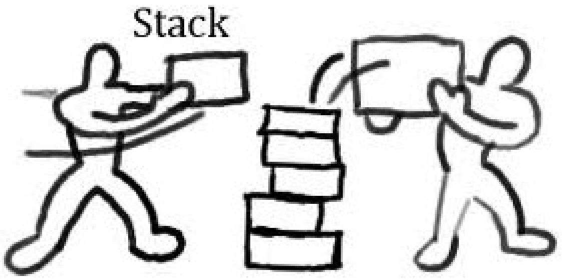
Make Files

- Macros in make files

```
EXE = myprog.exe
OBJS = main.o stack.o          <- defintion
CC = gcc
CFLAGS = -Wall -c
LFLAGS = -Wall

$(EXE): $(OBJS)           <- expansion
	$(CC) $(LFLAGS) $(OBJS) -o $(EXE)

main.o: main.c stack.h
	$(CC) $(CFLAGS) main.c

stack.o: stack.c stack.h
	$(CC) $(CFLAGS) stack.c

clean:
	rm *.o $(EXE)
```

Make Files

[DEMO: make file]



Code Optimization

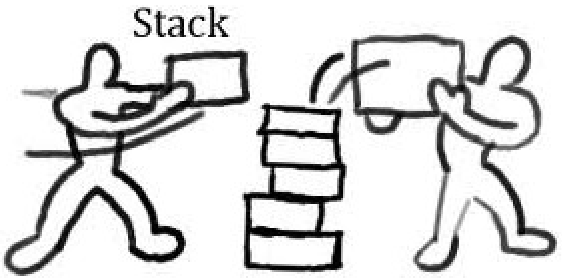
- Compiler optimizations
  - Gcc -O0 main.c
    - No code optimization at all
    - Typically used for compilation in 'debug' mode
  - Gcc -O1 main.c
    - Fast compilation with code optimization speed without increasing the code size
  - Gcc -O2 main.c
    - More code optimization for speed without increasing the code size but slower compilation
    - Often the 'best' choice for code deployment (compiling in 'release' mode)
  - Gcc -O3 main.c
    - Applies the most expensive code optimization rules for speed, but code size might increase
    - Example: function inlining
    - Because the potential increase of code size, the speed might even be slower (instruction cache)
  - Gcc -Os main.c
    - Optimize code for size

Code Optimization

- Use a code profiler
  - Profiler analyses a program by tracking which functions are called and how long each call takes
    - Find the most interesting pieces of code to optimize
  - Example: gprof
    - Build the program with '-pg' options
      - Example: gcc main.c -o myprog -pg
    - Run the program to collect profile data
      - Generates a file 'gmon.out'
    - Analyze profiler data using 'gprof'
      - Gprof myprog > myOutput

Code Optimization

[DEMO: gprof]



References

- For much more details on the utilities …
  - Check the man pages or manuals
    - Objdump, nm, ar, ldd, ldconfig
    - Gprof
      - http://sourceware.org/binutils/docs/gprof/index.html
    - Make
      - www.gnu.org/software/make/manual/make.html
- Background reading
    - Shared lib
      - www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html
    - Gcc optimization
      - www.linuxjournal.com/article/7269

LECTURE C7

Function Pointers and Data Abstraction

Function Types

```
typedef int Tfunc(float);

Or: typedef int (*Tfunc)(float);

int F( float x ) {
   /* do something */
}

int G( float y ) {
   /* do something else */
}
```

Function Pointers

```
// define function type variables
Tfunc *fun;

// assign a value to a function type variable
fun = &F;  // fun = &G;

// use the function type variable
x = (*fun)(1.2)
```

Function Pointers

```
// use a function type argument
float MoreFun( Tfunc *f ) {
   float x;
   x = (*f)( 1.2 );
   return x;
}

// function call
MoreFun( &F );   //or: MoreFun( &G )
```

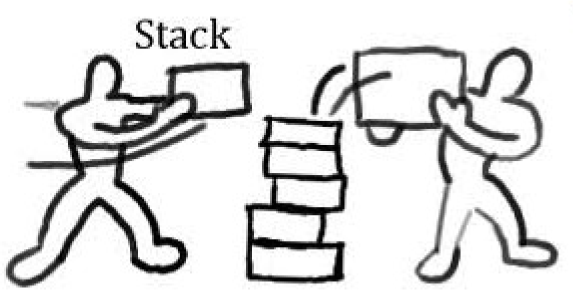
Function Pointers

- Example: use a callback error handler
- stack should be independent of ‘element’ data type
  - For ‘stack’ the element data type should be ‘void *’
  - Stack should use ‘callback-functions’ for operations on ‘elements’
    - E.g. CopyElement(...), DeleteElement(…), Equal(…), etc.

Function Pointers

- Example: use a callback function for error handling
- Study the code ...

[Toledo: Stack_V4E]



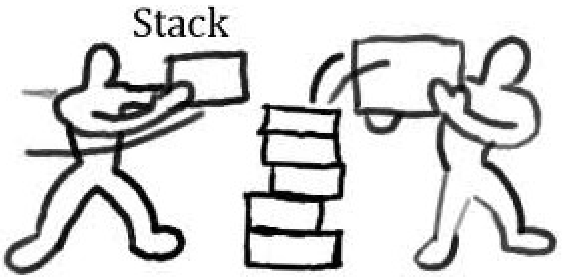
Function Pointers

- Example: stack should be independent of ‘element’ data type
  - For ‘stack’ the element data type should be ‘void *’
  - Stack should use ‘callback-functions’ for operations on ‘elements’
    - E.g. CopyElement(...), DeleteElement(...), Equal(...), etc.

Function Pointers

- Study the code ...

[Toledo: Stack_V8a]



Function Pointers

- STDLIB examples
  - Quicksort

    Void qsort (void *base, size_t n, size_t size,

    int (*cmp)(void *, void *)) ← *callback-function*

  - Binary search

    Void *bsearch (void *key, void *base, size_t n,

    size_t size, int (*cmp)(void *, void *)) ← *callback-function*

Program Termination

- Function pointers and atexit()
  - How to terminate a program?
  - Use ‘return <value>’ in main-func
  - Use ‘void abort(void)’
    - Abnormal termination of program!
    - Not really a ‘clean’ exit

Program Termination

- Function pointers and atexit()
  - How to terminate a program?

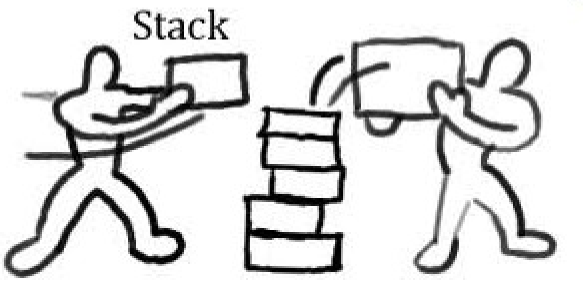
  - Void exit(int status)
    - Normal program termination
      - Status = EXIT_SUCCESS
      - Status = EXIT_FAILURE
    - Clean up is done
      - Files are flushed and closed, control is returned to the RTE, ...

Program Termination

- Function pointers and atexit()
  - How to terminate a program?

    - Int atexit(void (*func)(void))
      - Register function(s) that will be called after exit() or return()
      - Study the sample code …

[Toledo: AtExit]



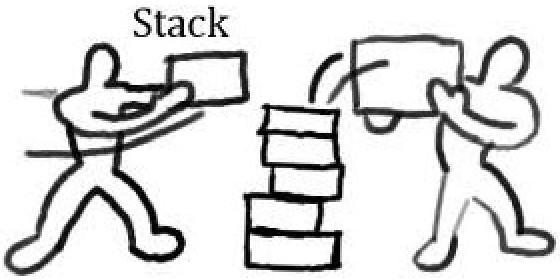
Data Hiding

- Stack user still has access to the stack type definition …
  - User can access/manipulate the data type without using the operators
  - Solution: “data hiding”
    - Use ‘void *’ to hide definition of stack
    - In ‘stack.h’: typedef void * Stack;

Data Hiding

- Study the code ...

[Toledo: Stack_V8b]



History and Standards

- Between 1969 and 1973
  - Denis Ritchie, Ken Thompson of AT&T Bell Labs rewrite an assembly implementation of Unix for the PDP-11 in a new language called C
  - One of the first OS kernels not implemented in assembly but a ‘higher’ programming language
  - C is derived from B
- C becomes a very popular programming language

History and Standards

- 1978: Brian Kernighan and Dennis Ritchie publish “The C Programming Language”
  - Served as the informal specification of the language: K&R C
- First C standard: C89
  - Released in 1989 by the ANSI X3J11 and ISO S22/WG14 C Standard Committees

History and Standards

- Second C standard: C99
  - Ratified in 1999
  - New features: complex numbers, variable length arrays, support for 64-bit computing, restricted pointers, …
  - C99 is backward compatible with C89
- Latest standard C11
  - Published Dec. 8th, 2011
  - New features: type generic macros, anonymous structures, improved Unicode support, atomic operations, multi-threading, and bounds-checked functions, improved compatibility with C++, …

Some Additional Features

- Not every compiler supports all C99/C11 features
  - Compile with argument '-std=c99' or '-std=c11'
- Compiler minimum resource limits
  - Support for identifiers with internal linkage of at least 63 chars (i.s.o 31 chars) and at least 31 chars for those with external linkage (i.s.o 6 chars)
  - Support for at least 1023 members in struct, union, enum
  - Support for at least 127 function parameters
  - ...

Some Additional Features

- Designated initializers for arrays, structs and unions

```
int a[5] = { [0] = 0, [2] = 10, [4] = 40 };

struct point {
   int x, y;
};

Struct point p = { .x = 20, .y = 30 };
```

Some Additional Features

- Type boolean, complex number, ...

```
#include <stdbool.h>

Bool stop = false; // true and false

While ( !stop )
{
   ...
}
```

Inline functions

- C99 standard allows inline functions
  - Compiler *may* replace every call to an inline functions with a copy of the function body
  - Only for functions with a short code body and that are called frequently
    - Inline avoids the use of the function stack which could improve performance

```
inline int minimum( int x, int y );

inline int minimum( int x, int y )
{
    return ( x < y ? x : y );
}
```

Some Additional Features

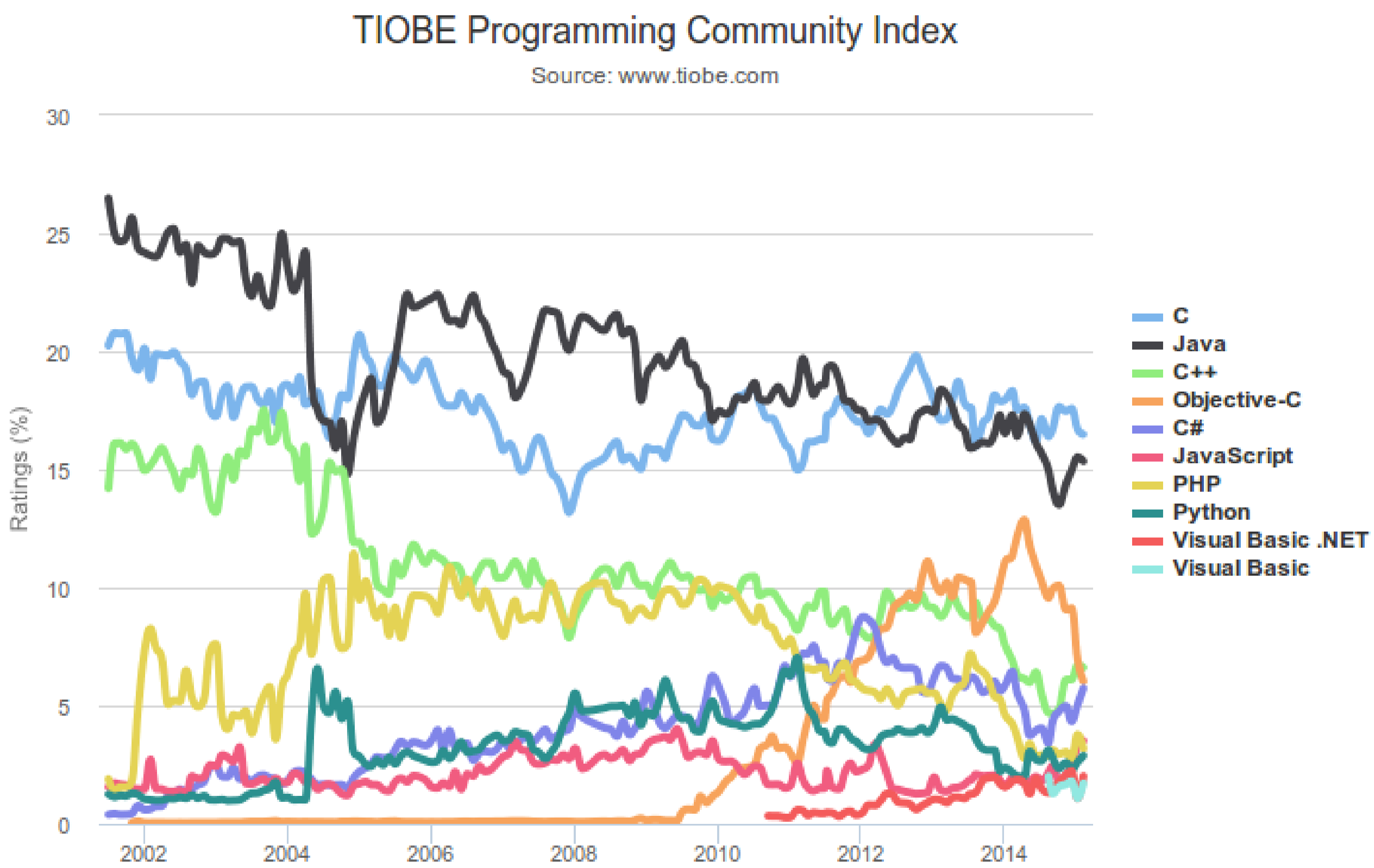
- Variable-length arrays
  - Arrays must have fixed size but C99 allows that the constant size is evaluated at run-time

```
int size;

...

printf(“Enter size of array: “);
scanf(“%d”, &size);
int data[size]; //define array 'data'
// changes to 'size' will not change the array size!

printf(“Array size in bytes = %d\n”, sizeof(data) );

...
```

OK, and why ...

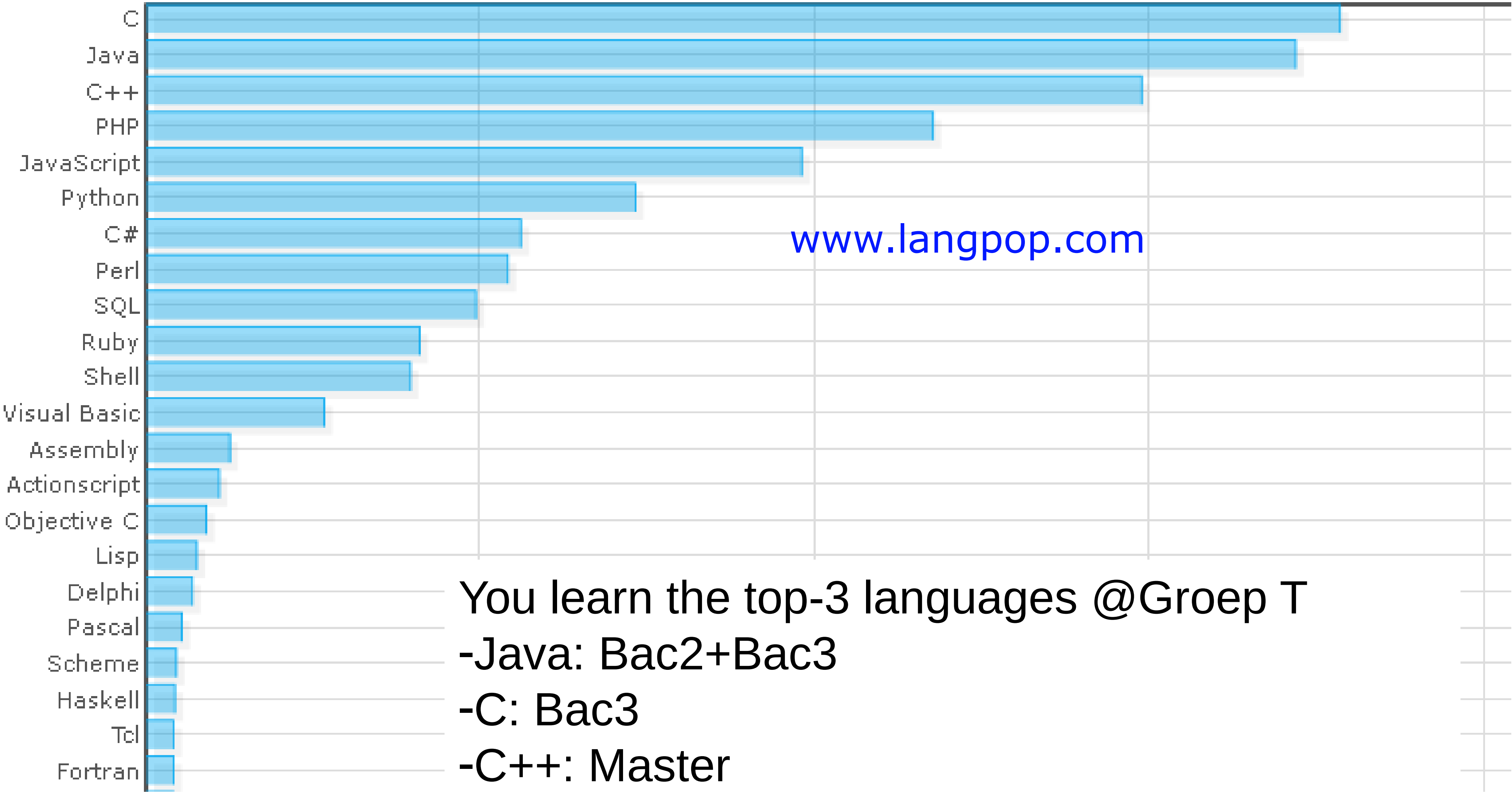
... did I need to learn all this?

C Is An Old Language And No Longer In Use! NOT TRUE



www.tiobe.com - Feb. 2015

C Is An Old Language And No Longer In Use! NOT TRUE



You learn the top-3 languages @Groep T

-Java: Bac2+Bac3

-C: Bac3

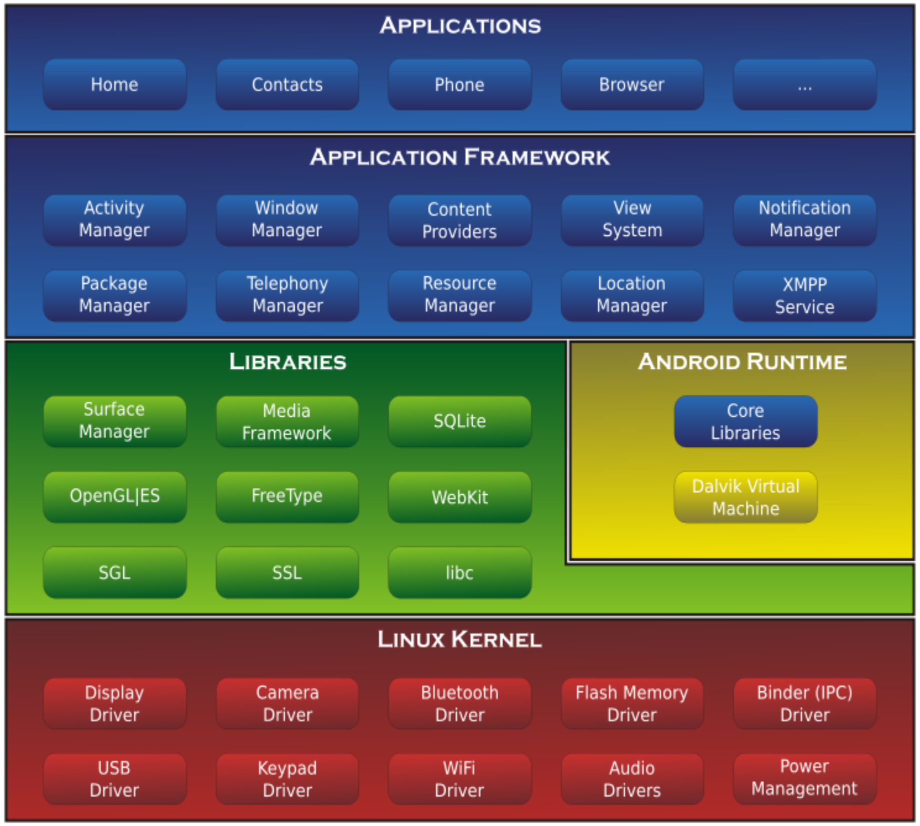
-C++: Master

C Is An Old Language And No Longer In Use! NOT TRUE

- C is the dominant language in embedded software
  - C compiler is one of the first compilers available on new computing platforms
  - C runs on every computing platform
    - PIC, Arduino, …
  - First choice for programming hardware and their interfaces
  - Embedded systems have typically limited computing and memory resources
    - That's the perfect habitat for C!

C Is An Old Language And No Longer In Use! NOT TRUE

- C is designed as 'system language' for OS, especially Unix/Linux
  - Large part of Linux, Android, Windows kernel is written in C
  - Java Virtual Machines are often written in C
  - Drivers are written in C
    - Only other alternative is assembly



C Is An Old Language And No Longer In Use! NOT TRUE

- Some famous application/tools/libraries written in C
  - MySQL, SQLite, ...
    - See source code
  - Apache web server
    - See source code
  - NginX (high-performance HTTP server powered by Netflix, GitHub, Zynga, WordPress, ...)
  - OpenSSL, Wireshark, ...
    - See source code
  - Google Big Table: C & C++
  - Matlab, GNU scientific library, Gnu multi-precision library, …
  - Compilers/interpreters for programming languages
    - PHP, Perl, Java, ...
  - See also: www.lextrait.com/vincent/implementations.html
- C is not the 'first choice' programming language for (GUI) applications
  - Go for C++, Java, Python, …
  - But: if speed matters, if resources are limited, … C can still be a valid choice

Other Reasons

- Other languages based on C
  - C++, Objective-C, Python, Java, C#, …
  - Google “Timeline of programming languages”

- Other programming languages are implemented in C
  - Python, Lua, …

Other Reasons

- C is an international standardized and non-proprietary programming language
  - C is highly portable
  - C is a supported and maintained by all major computing industry companies

- Read "**Ten Reasons to Teach and Learn Computer Programming in C",** H. Cheng [Toledo: Ten_reasons_to_learn_C.pdf]

Objectives Of This Course Part

- Be able to understand C source code.
- Be able to draw the memory layout (heap, details of the function stack, …) at an indicated instruction in a C program.
- Understand and be able to use the preprocessor for conditional compilation, header files, macros, ...
- Be able to design and implement software in C.
- Understand and be able to work with pointers in C. This includes, among others,
  - Pointers to implement call-by-reference parameter passing;
  - Pointers to implement dynamic memory and dynamic structures;
  - Function pointers
  - Void pointers for data hiding and abstraction
- Know and understand the C build process (preprocessor, compiler, linker, loader), be able to work with the gcc build utilities and with make files.
- Know and understand static/dynamic libraries and be able to work with libraries.
- Be able to use tools such as Gprof, objdump, Valgrind, ... to analyse and debug C code.
- Have basic knowledge on Linux and be able to work with the Linux command line (Bash shell, compiling and running programs, directory browsing, file copying, …).