C Compilation Process

Compiling a C program involves 3 separate tools

1. Pre-processor: Rewrites the code according to the defined *macros*
   - Lines begining with "#" are macros
   - `#define name value`: declare a sort of automatic search/replace
   - `#define name(params) value`: search/replace but with arguments
   - `#include "file"`: inline the content of the given file
   - `#ifdef name/#else/#endif`: mask parts of the file if `name` is defined
2. Compiler: Translates the code into assembly
3. Linker: Take elements in assembly and resolve library dependencies
   - If your code uses function `cos()`, you need the math lib
   - The linker solves a puzzle to ensure that every used function get defined

This process is rather transparent to the user

- You edit your code (in emacs/vi/eclipse)
- You launch gcc, which lauches mandatory tools automatically
- You mainly need to know that when you get error messages

The C preprocessor

Motivation

- C designed to work at (very) low level on a variety of machines
  Sometimes, the only way to portability for a given function is:
  Have several versions (windows, linux, mac); pick the right one at compilation
- C is a very old language $\leadsto$ we sometimes want to *extend* it a bit

The C preprocessor: in direct line with Paleolithic

- I'm not sure you'll ever have to use such a rudimentary tool
- It's as dumb as possible, but it perfectly fulfills its tasks
- It's even sometimes used outside of the C ecosystem
- Beware, that's the perfect tool to make your code unreadable

Preprocessor: Macros without Arguments

`#define MACRO_NAME value`

- This requests a find/replace
  Ex: `#define MAX 12` $\leadsto$ change every “MAX” into 12
- Numerical constants must be defined that way (or const variables, or enums)
- Always write macro names in all upper case (to make clear what they are)
- Preprocessor lines expect no final semi-column (“;”)
- Always put too much parenthesis. Think of the result of:

```
#define TWELVE 10+2
int x = TWELVE * 2; //⇝ x equals 10+2*2 = 14, not 12*2=24
// #define TWELVE (10+2) would fix it
```

- Preprocessor directive must be on one line only $\leadsto$ escape return carriages

```
#define MY_MACRO this is \
                   a multi-line \
                   macro definition
```

More on Preprocessor Macros

Predefined macros

- __STDC__: 1 if the compiler conforms to ANSI C
- __FILE__: current file; __LINE__: current line in that file

⇝ `printf("%s:%d: was here\n", __FILE__, __LINE__);`

`#define MACRO_NAME(parameters) value`

- Programmable find/replace

  Ex: `#define MAX(a,b) ((a)>(b)?(a):(b))` (yep, there is no max() in C)

`#undef MACRO`

- Forget previous definition of this macro

`#include <header file>`

- As previously said, line replaced by whole content of file
- Header files are source file intended to be loaded this way

Conditional compilation with the preprocessor

Conditional on macro definitions

```
#ifdef macro_name
  /* Code to use if macro defined */
#else
  /* Code to use otherwise */
#endif

#ifndef macro_name
  /* Code if macro not defined */
#else
  /* Code if defined */
#endif
```

Conditional on expressions

```
#if constant_expression1
  /* some C code */
#elif constant_expression2
  /* some C code */
#else
  /* some C code */
#endif

#if 0
  code to kill
#endif
```

Protect against multiple inclusions

```
/* mainly useful for header files */
#ifndef SOME_UNIQUE_NAME
#define SOME_UNIQUE_NAME
   ...
#endif
```

Redefine a macro

```
#ifdef MACRO
#undef MACRO
#endif
#define MACRO blabla
```

`#error "biiiirk"`

- Raises a compilation error with given message (yep, that's sometimes useful)

What if you get error messages when compiling

Some examples

- `foo.c:71:2: error: invalid preprocessing directive #deifne`
  The preprocessor is not happy: check file `foo.c`, line 71, column 2
- `foo.c:72: error: expected ')' before 'char'`
  Compiler's not happy (syntax error)
- `foo.c:74: error: redefinition of 'myFunc'`
  `foo.c:72: error: previous definition of 'myFunc' was here`
  Defining the same function twice makes the linker unhappy
- `/usr/lib/crt1.o: In function '_start':`
  `(.text+0x18): undefined reference to 'main'`
  `collect2: ld returned 1 exit status`
  A function is used, but never defined
  (see RS lecture next year to understand the detail of the message)
- `Segmentation fault ./myProg`
  Your program messed up the memory (valgrind knows where)

How to react when you get error messages (and you will)

- Don't panic, even if the message seem cryptic (they often are)
- Read the message: they are sometimes even understandable
- Don't even read the **second** message: the parser often gets lost after first error

Strings in C

Unfortunately, there is no standard type in C to describe strings...

- Instead, the C idiomatic is to use arrays of chars
- In turn, arrays are unpleasant because they do not contain their own length
- So **by convention** every C string should be zero-terminated
  i.e. the last value in the array is the special char '\0' (different from '0')
- Beware, to store a string of 5 letters, you need 6 positions:

```c
char str[6]="hello";
```

h	e	l	l	o	\0

- Useful functions for such strings: `strlen()`, `strcpy()`, `strcmp()`, ...
- But you are free to not follow that convention if you prefer to do otherwise (you just have to do it all by yourself then)
- If the size is given elsewhere, you can use `char *str;` for `char str[5];`
  (MUCH more to come on that little * sign)
- Don't mix the char `'a'` with the string `"a"`

Structures in C

This is a fundamental construction in C

- Group differing aspects of a given concept, just like Java objects
  Vocabulary: We speak of **structure members** and **object fields**
- But they (usually) don't contain the associated methods/functions

Definition example

```
struct point {
  double x;
  double y;
  int rank;
}; // beware of the trailing ;
```

Usage example

```
struct point p1; // the type name is ``struct point''
p1.x = 4.2;
p1.y = 3.14;
p1.rank = 1;
struct point p2 = { 4.2, 3.14, 2 };
```

Structures as parameter and return values

```
struct point translate(struct point p,
                       double dx, double dy) {
  struct point res = p;
  res.x += dx;
  res.y += dy;
  return res;
}
```

Declare and use at once

```
struct point {
  int x;
} p1,p2; // variables of that type
```

```
struct { // Anonymous structure
  int x;
} p1,p2; // variables of that type
```

- Parameter and return values are *copied* (no border effect; inherent inefficiency)
- Remarks: Members can be `structs` too; No global operators (such as ==)

Enumerations in C

Basics

- They are used to group **values** of the same lexical scope
- A variable of type *color* can take a value within {blue, red, white, yellow}

Definition example

```
enum color {
  blue, red, white, yellow
}; // beware of the trailing ;
```

Usage example

```
enum color bikesheld; // the type name is ``enum color''
bikesheld = white;
```

Enumerations can be used as parameter and return values

```
enum color make_white(enum color c) {
  return white; // Yes, this function is useless as is...
}
```

- Main advantage: there is a compilation error if you forget a value in a switch (instead of silently ignoring the whole block when the case occurs, which is a pain)
- Every arithmetic and logical operators can be used (white+1$\leadsto$yellow)

Java enums

- They exist in Java, too. Much more powerful and complicated. Rarely used.

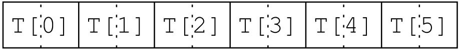
Memory layout of C type constructors

Impossible to master C without understanding the memory layout

- (This is because memory is a kind of unsorted magma in C)
- **First absolute rule:** successive elements are stored in order in memory

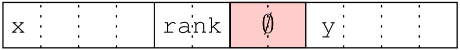
```
struct point {
  double x;
  double y;
  int rank;    };
```


```
int T[6];
```



- But the compiler is free to add padding space to respect alignment constraints

```
struct point {
  double x;
  int rank;
  double y;
};
```



- Compiler-dependent/processor-dependent, so you can hardly rely on it...

Type aliasing in C

Motivation

- Type names quickly become quite long: `enum color`,
- Variable square being an array of four points: `struct point square[4]`

⇒ Keyword `typedef` used to declare **type aliases**

Usage

- Reading a `typedef`: "the last word is an alias for everything else on the line"

Basic example

```
struct point {
  double x;
  double y;
};
typedef struct point point_t;
...
point_t p;
p.x = 4.2;
p.y = 3.14;
```

All-in-one example

```
typedef struct point {
  double x, y;
} point_t;
```

Complex example

```
typedef point_t square_t[4];
square_t s;      s[0].x=3.14;
```

- typedefs are mandatory to organize your code...
- ...but can easily be misused to make your code messy and unreadable (just like about every C idiomatic constructs)

Writing to the `stdout` with the `printf` function

Naive usage

- Write the fixed string "hello" to the terminal: `printf("hello")`
- Write that string and return to the line beginning: `printf("hello\n")`

Writing to the `stdout` with the `printf` function

Naive usage

- Write the fixed string "hello" to the terminal: `printf("hello")`
- Write that string and return to the line beginning: `printf("hello\n")`

Basic usage

- To output variables, put place holders in the format string:

```
int x=3; printf("value: %d\n",x)
```

- Use several place holders to display several variables:

```
int x=3; int y=2; printf("x: %d; y: %d\n",x,y)
```

- The kind of place holder gives the type of variable to display

%d	integer (decimal)
%f	floating point number
%c	char
%s	string (nul-terminated char array)
%%	the % char

- If you use the wrong conversion specifier, strange things will happen including a brutal ending of your program – SEGFAULT

Advanced `printf` usage

Other conversion specifiers

%u	unsigned integer
%ld	long integer
%lu	long unsigned integer
%o	integer to display in octal
%x	integer to display in hexadecimal
%e	floating point number to display in scientific notation

Formating directive modifiers

- You can specify that you want to see at least 3 digits: `printf("%3d",x);`
- Or that you want exactly 2 digits after the dot: `printf("%.2d",x);`
- Or both at the same time: `printf("%3.2f",x);`
- Or that the output must be 0-padded: `printf("%03.2f",x);` $\rightsquigarrow$ 003.14
- Or that you want to see at most 3 chars: `printf("%.3s",str);`

Many other options exist, full list in `man 3 printf`

Reading from `stdin` with the `scanf` function

Works quite similarly to `printf`, but...

- Read an integer: `int x; scanf("%d", &x);`
- Read a double: `double d; scanf("%f", &d);`
- Read a char: `char c; scanf("%c", &c);`
- Read a string: `char str[120]; scanf("%c", str);`
- Read two things: `int x;char c; scanf("%d%c", &x, &c);`

So...

- You need to add a little & to the variable...
- ...unless when the variable is a string (we'll explain later why)
- Format string can contain other chars than converters: they **must** be in input
- A space in format will match any amount of white chars (spaces, \n, tabs)
- Note that scanf returns the amount of chars it managed to read
  Useful for error checking: what if that's not an integer but something else?

File I/O

#include <stdio.h>

printf/scanf functions have nice friends for that

- Writing to `stderr`: `fprintf(stderr,"warning\n")`
  - `fprintf` works just like printf, taking a file handler as first argument
  - Likewise `fscanf` is just like `scanf`, with a handler as first argument
- Declaring a file handler (a variable describing a file): `FILE* handler;`
- Opening a file for reading `handler = fopen("myfile","r");`
- Opening a file for writing `handler = fopen("myfile","w");`
- Opening a file in read/write mode `handler = fopen("myfile","r+");`
- Checking that the opening went well: `if (handler==NULL) {problem}`
- Checking whether we reached the end of file `if (feof(handler)) {done}`
- Closing a file: `fclose(handler);`

In UNIX, everything is a file, and it makes things easier

Variables in C

Kind of identifiers in C

- Little difference between variables and functions: they are all **identifiers**
- Every C identifiers can be either **global** or **local**
- Main differences: **scope** (visibility) and **lifetime**

Local Identifiers

- They are declared within a function
- Side note: nested functions are forbidden in standard C
  gcc allows nested functions as a language extension – I recommend not using them
- **Scope:** Usable from the block where they are declared
- **Lifetime:** Valid only until the execution leaves the block

Global identifiers

- They are declared outside of any function
- **Scope:** Usable from the whole project
- **Lifetime:** permanent
- (yes, there is no such thing in Java)

First Weird Code with Variables

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

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a $\boxed{}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int b=0;
18:       printf("a: %d, b: %d\n", a,b);
19:     }
20:     printf("a: %d, b: %d\n", a,b);
21:   }
22:   printf("a: %d, b: %d\n",a,b);
23:   return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 *a* ☐

l3 *b* ☐

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 *a* $\boxed{0}$

l3 *b* $\boxed{}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 *a* $\boxed{0}$

l3 *b* $\boxed{0}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 *a* $\boxed{0}$

l3 *b* $\boxed{0}$

l5 `a: 0; b: 0`

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 *a* $\boxed{0}$

l3 *b* $\boxed{0}$

l5 `a: 0; b: 0`

l6 *a* $\boxed{5}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

l13 b_{13} $\boxed{10}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

l13 b_{13} $\boxed{10}$

l14 `a: 10; b: 10`

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

l13 b_{13} $\boxed{10}$

l14 `a: 10; b: 10`

l15 b_{13} $\boxed{15}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int b=0;
18:       printf("a: %d, b: %d\n", a,b);
19:     }
20:     printf("a: %d, b: %d\n", a,b);
21:   }
22:   printf("a: %d, b: %d\n",a,b);
23:   return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

l13 b_{13} $\boxed{10}$

l14 `a: 10; b: 10`

l15 b_{13} $\boxed{15}$

l17 b_{17} $\boxed{0}$

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

l13 b_{13} $\boxed{10}$

l14 `a: 10; b: 10`

l15 b_{13} $\boxed{15}$

l17 b_{17} $\boxed{0}$

l18 `a: 10; b: 0`

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

l13 b_{13} $\boxed{10}$

l14 `a: 10; b: 10`

l15 b_{13} $\boxed{15}$

l17 b_{17} $\boxed{0}$

l18 `a: 10; b: 0`

l20 `a: 10; b: 15`

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

l13 b_{13} $\boxed{10}$

l14 `a: 10; b: 10`

l15 b_{13} $\boxed{15}$

l17 b_{17} $\boxed{0}$

l18 `a: 10; b: 0`

l20 `a: 10; b: 15`

l22 `a: 10; b: 0`

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1 a_1 $\boxed{0}$

l3 b_3 $\boxed{0}$

l5 `a: 0; b: 0`

l6 a_1 $\boxed{5}$

l8 a_8 $\boxed{??}$

l9 `a: ??; b: 0`

l11 a_1 $\boxed{10}$

l13 b_{13} $\boxed{10}$

l14 `a: 10; b: 10`

l15 b_{13} $\boxed{15}$

l17 b_{17} $\boxed{0}$

l18 `a: 10; b: 0`

l20 `a: 10; b: 15`

l22 `a: 10; b: 0`

Ok, but how to **understand** it?

First Weird Code with Variables

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Remarks

- Yes, we can use anonymous blocks
- We can declare variables in there
- We can override variables this way
- All this is possible in Java too!

What does this code do?

l1	a_1 $\boxed{0}$	l13	b_{13} $\boxed{10}$
l3	b_3 $\boxed{0}$	l14	`a: 10; b: 10`
l5	`a: 0; b: 0`	l15	b_{13} $\boxed{15}$
l6	a_1 $\boxed{5}$	l17	b_{17} $\boxed{0}$
l8	a_8 $\boxed{??}$	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l11	a_1 $\boxed{10}$	l22	`a: 10; b: 0`

Ok, but how to **understand** it?

- Think of a stack containing locals

Variables are Stored on a Stack

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones

Variables are Stored on a Stack

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones

Variables are Stored on a Stack

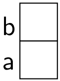
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

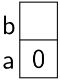
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones

b	0
a	0

Variables are Stored on a Stack

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	a: 0; b: 0	l18	a: 10; b: 0
l9	a: ??; b: 0	l20	a: 10; b: 15
l14	a: 10; b: 10	l22	a: 10; b: 0

The stack over time

- Higher variables mask deeper ones

b	0
a	0

l5

Variables are Stored on a Stack

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones

	l5		
b	0	b	0
a	0	a	5

Variables are Stored on a Stack

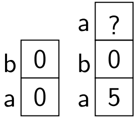
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5 `a: 0; b: 0`
l9 `a: ??; b: 0`
l14 `a: 10; b: 10`
l18 `a: 10; b: 0`
l20 `a: 10; b: 15`
l22 `a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



l5

Variables are Stored on a Stack

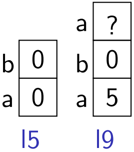
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

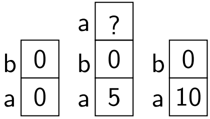
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



l5 l9

Variables are Stored on a Stack

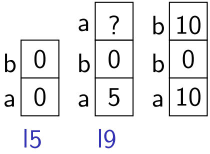
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

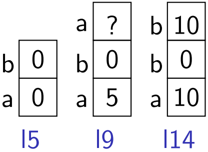
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

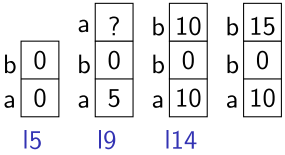
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

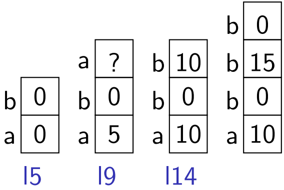
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

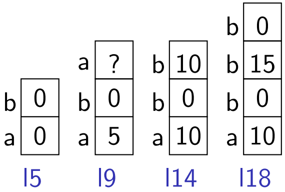
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

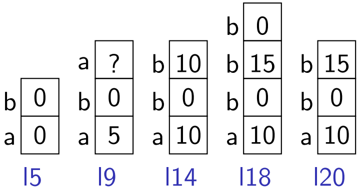
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5	`a: 0; b: 0`	l18	`a: 10; b: 0`
l9	`a: ??; b: 0`	l20	`a: 10; b: 15`
l14	`a: 10; b: 10`	l22	`a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones



Variables are Stored on a Stack

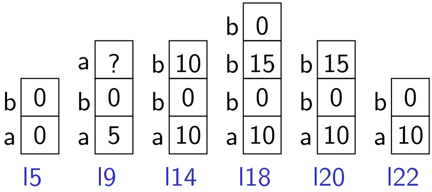
```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printf("a: %d, b: %d\n",a,b);
 6:  a += 5;
 7:  {
 8:    int a;
 9:    printf("a: %d, b: %d\n",a,b);
10:  }
11:  a += 5;
12:  {
13:    int b=a;
14:    printf("a: %d, b: %d\n",a,b);
15:    b += 5;
16:    {
17:      int b=0;
18:      printf("a: %d, b: %d\n", a,b);
19:    }
20:    printf("a: %d, b: %d\n", a,b);
21:  }
22:  printf("a: %d, b: %d\n",a,b);
23:  return 0;
24:}
```

Explaining the outputs

l5 `a: 0; b: 0`
l9 `a: ??; b: 0`
l14 `a: 10; b: 10`
l18 `a: 10; b: 0`
l20 `a: 10; b: 15`
l22 `a: 10; b: 0`

The stack over time

- Higher variables mask deeper ones

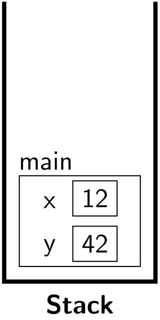


Function Parameters

Parameters are stacked too

- One **stack frame** per function (containing local vars and parameters)
- Stack frame: created on function call, destructed when the function returns
- Parameters can be seen as local variables (can even be modified)
- Parameters are **passed by value** (ie, copied over)

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

int main() {
  int x=12;
  int y=42;
  return max(x,y);
}
```

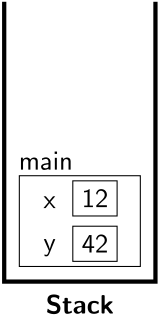


Function Parameters

Parameters are stacked too

- One **stack frame** per function (containing local vars and parameters)
- Stack frame: created on function call, destructed when the function returns
- Parameters can be seen as local variables (can even be modified)
- Parameters are **passed by value** (ie, copied over)

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

int main() {
  int x=12;
  int y=42;
  return max(x,y);
}
```

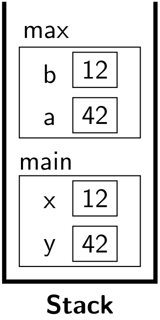


Function Parameters

Parameters are stacked too

- One **stack frame** per function (containing local vars and parameters)
- Stack frame: created on function call, destructed when the function returns
- Parameters can be seen as local variables (can even be modified)
- Parameters are **passed by value** (ie, copied over)

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

int main() {
  int x=12;
  int y=42;
  return max(x,y);
}
```

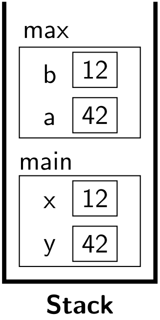


Function Parameters

Parameters are stacked too

- One **stack frame** per function (containing local vars and parameters)
- Stack frame: created on function call, destructed when the function returns
- Parameters can be seen as local variables (can even be modified)
- Parameters are **passed by value** (ie, copied over)

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

int main() {
  int x=12;
  int y=42;
  return max(x,y);
}
```

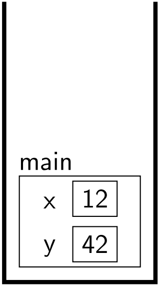


Function Parameters

Parameters are stacked too

- One **stack frame** per function (containing local vars and parameters)
- Stack frame: created on function call, destructed when the function returns
- Parameters can be seen as local variables (can even be modified)
- Parameters are **passed by value** (ie, copied over)

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

int main() {
  int x=12;
  int y=42;
  return max(x,y);
}
```



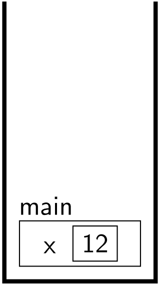
Stack

Function Parameters Tricks

Parameters are passed by value

- We just said that but it is not as natural as it seems
- It forbids any side effects on parameters

```c
void triple(int a) {
  a=a*3;
  return;
}

int main() {
  int x=12;
  triple(x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

```
$ myprog
```

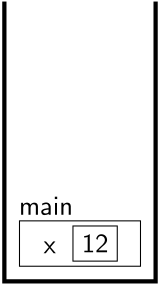
Output

Function Parameters Tricks

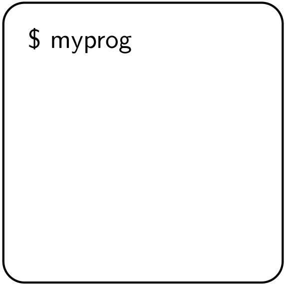
Parameters are passed by value

- We just said that but it is not as natural as it seems
- It forbids any side effects on parameters

```c
void triple(int a) {
  a=a*3;
  return;
}

int main() {
  int x=12;
  triple(x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack



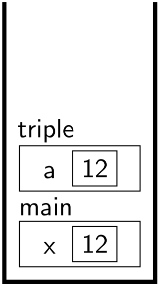
Output

Function Parameters Tricks

Parameters are passed by value

- We just said that but it is not as natural as it seems
- It forbids any side effects on parameters

```c
void triple(int a) {
  a=a*3;
  return;
}

int main() {
  int x=12;
  triple(x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

```
$ myprog
```

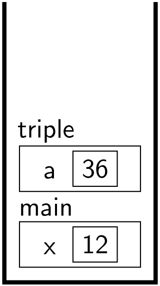
Output

Function Parameters Tricks

Parameters are passed by value

- We just said that but it is not as natural as it seems
- It forbids any side effects on parameters

```
void triple(int a) {
  a=a*3;
  return;
}

int main() {
  int x=12;
  triple(x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

```
$ myprog
```

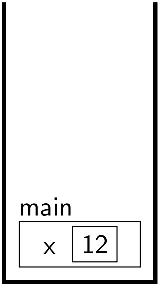
Output

Function Parameters Tricks

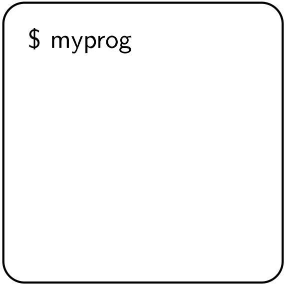
Parameters are passed by value

- We just said that but it is not as natural as it seems
- It forbids any side effects on parameters

```
void triple(int a) {
  a=a*3;
  return;
}

int main() {
  int x=12;
  triple(x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack



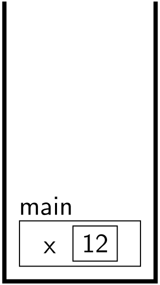
Output

Function Parameters Tricks

Parameters are passed by value

- We just said that but it is not as natural as it seems
- It forbids any side effects on parameters

```c
void triple(int a) {
  a=a*3;
  return;
}

int main() {
  int x=12;
  triple(x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

```
$ myprog
x: 12
```

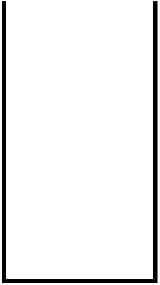
Output

Function Parameters Tricks

Parameters are passed by value

- We just said that but it is not as natural as it seems
- It forbids any side effects on parameters
- There is no way to avoid passing by value
- But pointers help: scanf manages to “modify its arguments”

```c
void triple(int a) {
  a=a*3;
  return;
}

int main() {
  int x=12;
  triple(x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

```
$ myprog
x: 12
$
```

Output

Weird Code with Function Calls

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

Code similar to previously

- Call printab() for display, not printf()

Weird Code with Function Calls

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printab(a,b);
 6:  a += 5;
 7:  { int a;
 8:    printab(a,b);
 9:  }
10:  a += 5;
11:  { int b=a;
12:    printab(a,b);
13:    b += 5;
14:    { int b=0;
15:      printab(a,b);
16:    }
17:    printab(a,b);
18:  }
19:  printab(a,b);
20:  return 0;
21:}
22:int printab(int b, int a) {
23:  printf("a:%d, b:%d\n",a,b);
24:}
```

Code similar to previously

- Call printab() for display, not printf()

Old Output

l5	a: 0; b: 0	l18	a: 10; b: 0
l9	a: ??; b: 0	l20	a: 10; b: 15
l14	a: 10; b: 10	l22	a: 10; b: 0

New Output

l5	a:0; b:0	l18	a:0; b:10
l9	a:0; b:??	l20	a:15; b:10
l14	a:10; b:10	l22	a:0; b:10

This is all inverted!

The trick comes from...

Weird Code with Function Calls

```
 1:int a;
 2:int main() {
 3:  int b;
 3:  a=0;
 4:  b=a;
 5:  printab(a,b);
 6:  a += 5;
 7:  { int a;
 8:    printab(a,b);
 9:  }
10:  a += 5;
11:  { int b=a;
12:    printab(a,b);
13:    b += 5;
14:    { int b=0;
15:      printab(a,b);
16:    }
17:    printab(a,b);
18:  }
19:  printab(a,b);
20:  return 0;
21:}
22:int printab(int b, int a) {
23:  printf("a:%d, b:%d\n",a,b);
24:}
```

Code similar to previously

- Call printab() for display, not printf()

Old Output

l5	a: 0; b: 0	l18	a: 10; b: 0
l9	a: ??; b: 0	l20	a: 10; b: 15
l14	a: 10; b: 10	l22	a: 10; b: 0

New Output

l5	a:0; b:0	l18	a:0; b:10
l9	a:0; b:??	l20	a:15; b:10
l14	a:10; b:10	l22	a:0; b:10

This is all inverted!

The trick comes from. . .

- printab's parameters, which are inverted

The keyword static

This little keyword has two (quite differing) meanings

When applied to global identifiers

- Reduces visibility: from "the whole project" to "this file" (as if it were local)
- Lifetime remains unchanged
- Java equivalent: private

When applied to local identifiers

- Increases lifetime: from "for this call" to "for ever" (as if it were global)
- Visibility remains unchanged
- Similar concept in Java: static

The keyword static

This little keyword has two (quite differing) meanings

When applied to global identifiers

- Reduces visibility: from "the whole project" to "this file" (as if it were local)
- Lifetime remains unchanged
- Java equivalent: private

When applied to local identifiers

- Increases lifetime: from "for this call" to "for ever" (as if it were global)
- Visibility remains unchanged
- Similar concept in Java: static

	Visibility	Lifetime
Functions	Whole Project	For Ever
Global Variable	Whole Project	For Ever
Local Variable	Current Block	Until End of Block

The keyword static

This little keyword has two (quite differing) meanings

When applied to global identifiers

- Reduces visibility: from "the whole project" to "this file" (as if it were local)
- Lifetime remains unchanged
- Java equivalent: private

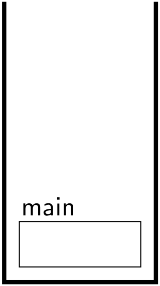
When applied to local identifiers

- Increases lifetime: from "for this call" to "for ever" (as if it were global)
- Visibility remains unchanged
- Similar concept in Java: static

	Visibility	Lifetime
Functions	Whole Project	For Ever
Global Variable	Whole Project	For Ever
Static Global Variable	This File Only	For Ever
Static Local Variable	Current Block	For Ever
Local Variable	Current Block	Until End of Block

More on Static Local Variables

```c
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



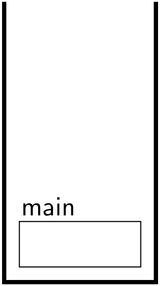
Stack

```
$ myprog
```

Output

More on Static Local Variables

```
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



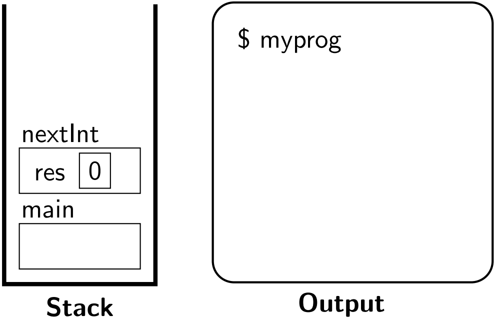
Stack

```
$ myprog
```

Output

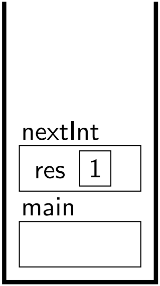
More on Static Local Variables

```c
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



More on Static Local Variables

```
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



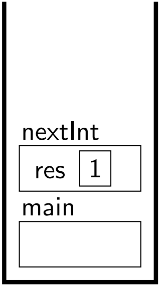
Stack

```
$ myprog
```

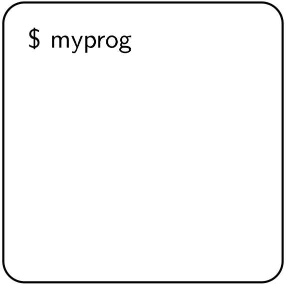
Output

More on Static Local Variables

```c
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



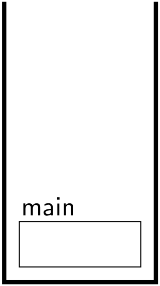
Stack



Output

More on Static Local Variables

```
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



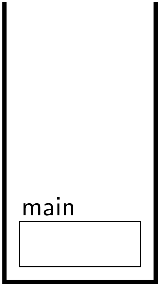
Stack

```
$ myprog
next: 1
```

Output

More on Static Local Variables

```c
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



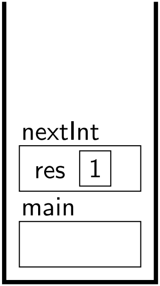
Stack

```
$ myprog
next: 1
```

Output

More on Static Local Variables

```c
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



Stack

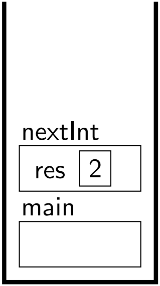
```
$ myprog
next: 1
```

Output

- The value remains from one call to another (initializer evaluated only once)

More on Static Local Variables

```
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



Stack

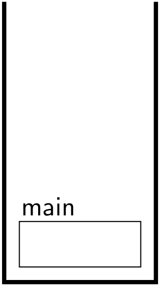
```
$ myprog
next: 1
```

Output

- The value remains from one call to another (initializer evaluated only once)

More on Static Local Variables

```c
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



Stack

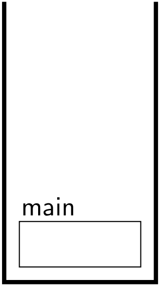
```
$ myprog
next: 1
```

Output

- The value remains from one call to another (initializer evaluated only once)

More on Static Local Variables

```c
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



Stack

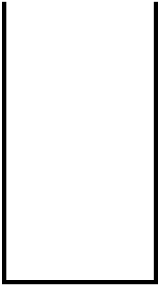
```
$ myprog
next: 1
next: 2
```

Output

- The value remains from one call to another (initializer evaluated only once)

More on Static Local Variables

```c
int nextInt() {
  static int res=0;
  res+=1;
  return res;
}

int main() {
  printf("next:%d",nextInt());
  printf("next:%d",nextInt());
  return EXIT_SUCCESS;
}
```



Stack

```
$ myprog
next: 1
next: 2
$
```

Output

- The value remains from one call to another (initializer evaluated only once)
- This variable cannot live on the stack: would have been erased by another call
- Understanding where it lives require some more background on the system (actually, the globals are not on the stack either)

Processes Memory Layout

Primer from Next Year in System Course

- The memory of each process is split in 3 big **segments**
- Heap is for the manually managed memory (see in half an hour)

- This is a simplification, but the ideas are there

Data	**Heap**		**Stack**
Code+Globals	Dynamic Memory		Stack Frames

Processes Memory Layout

Primer from Next Year in System Course

- The memory of each process is split in 3 big **segments**
- Heap is for the manually managed memory (see in half an hour)
- If more stack frames needed, the size of the stack grows toward the heap
  Conversely, the heap can grow toward the stack

- This is a simplification, but the ideas are there

Processes Memory Layout

Primer from Next Year in System Course

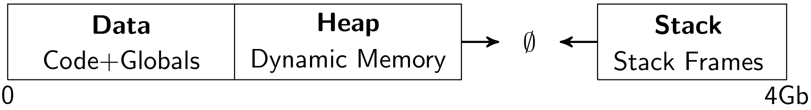
- The memory of each process is split in 3 big **segments**
- Heap is for the manually managed memory (see in half an hour)
- If more stack frames needed, the size of the stack grows toward the heap
  Conversely, the heap can grow toward the stack
- Between Heap and Stack, there is a hole
- If that hole becomes full (stack reaches heap), the process runs out of memory
- This is a simplification, but the ideas are there

Data	**Heap**		**Stack**
Code+Globals	Dynamic Memory	$\rightarrow \emptyset \leftarrow$	Stack Frames

Processes Memory Layout

Primer from Next Year in System Course

- The memory of each process is split in 3 big **segments**
- Heap is for the manually managed memory (see in half an hour)
- If more stack frames needed, the size of the stack grows toward the heap
  Conversely, the heap can grow toward the stack
- Between Heap and Stack, there is a hole in the addressing space
- If that hole becomes full (stack reaches heap), the process runs out of memory
- This is a simplification, but the ideas are there



Processes Memory Layout

Primer from Next Year in System Course

- The memory of each process is split in 3 big **segments**
- Heap is for the manually managed memory (see in half an hour)
- If more stack frames needed, the size of the stack grows toward the heap
  Conversely, the heap can grow toward the stack
- Between Heap and Stack, there is a hole in the addressing space
- If that hole becomes full (stack reaches heap), the process runs out of memory
- This is a simplification, but the ideas are there

Data	**Heap**				**Stack**
Code+Globals	Dynamic Memory	$\rightarrow$	$\emptyset$	$\leftarrow$	Stack Frames

0 ... 4Gb

Where do symbols live?

- Functions: in Data segment
- Globals: in Data segment
- Locals: in Stack segment
- Static Locals: in Data segment
  (just like globals!)

More on Memory

Solving the Enigma of Static Locals Storage raises New Questions

- What is the addressing space?
- How to get a valid mental representation of the memory?

- What is an address?
- Why the stack bottom at 4Gb?
- Where is my stack if my laptop does not have 4Gb?

More on Memory

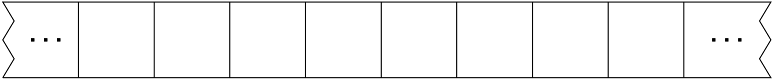
Solving the Enigma of Static Locals Storage raises New Questions

- What is the addressing space? $\leadsto$ This is another name for "memory"
- How to get a valid mental representation of the memory?
- What is an address?
- Why the stack bottom at 4Gb?
- Where is my stack if my laptop does not have 4Gb?

More on Memory

Solving the Enigma of Static Locals Storage raises New Questions

- What is the addressing space? $\leadsto$ This is another name for "memory"
- How to get a valid mental representation of the memory?
  $\leadsto$ Think of a very large array of cells. Each cell is 1 byte (8 bits) wide.



- What is an address?
- Why the stack bottom at 4Gb?
- Where is my stack if my laptop does not have 4Gb?

More on Memory

Solving the Enigma of Static Locals Storage raises New Questions

- What is the addressing space? $\leadsto$ This is another name for "memory"
- How to get a valid mental representation of the memory?
  $\leadsto$ Think of a very large array of cells. Each cell is 1 byte (8 bits) wide.

...	42	43	44	45	46	47	48	49	...

- What is an address? $\leadsto$ Memory cells are numbered.
  The **address** of a given memory cell is its number in rank
- Why the stack bottom at 4Gb?
- Where is my stack if my laptop does not have 4Gb?

More on Memory

Solving the Enigma of Static Locals Storage raises New Questions

- What is the addressing space? $\leadsto$ This is another name for "memory"
- How to get a valid mental representation of the memory?
  $\leadsto$ Think of a very large array of cells. Each cell is 1 byte (8 bits) wide.

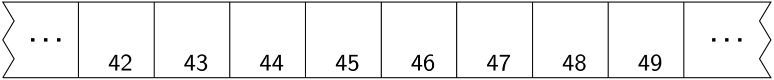
...	42	43	44	45	46	47	48	49	...

- What is an address? $\leadsto$ Memory cells are numbered.
  The **address** of a given memory cell is its number in rank
- Why the stack bottom at 4Gb? $\leadsto$ Because this is MAXINT on 32bits
  And the picture supposed that we were in 32bits for simplicity sake.
- Where is my stack if my laptop does not have 4Gb?

More on Memory

Solving the Enigma of Static Locals Storage raises New Questions

- What is the addressing space? $\leadsto$ This is another name for "memory"
- How to get a valid mental representation of the memory?
  $\leadsto$ Think of a very large array of cells. Each cell is 1 byte (8 bits) wide.



- What is an address? $\leadsto$ Memory cells are numbered.
  The **address** of a given memory cell is its number in rank
- Why the stack bottom at 4Gb? $\leadsto$ Because this is MAXINT on 32bits
  And the picture supposed that we were in 32bits for simplicity sake.
- Where is my stack if my laptop does not have 4Gb?
  - Within the process, we are speaking of *virtual addresses*
  - They get converted into *physical ones* by the OS
  - But this all is to be seen in RSA (not even RS – end of next year)

Storing Data in Memory

What can get stored in a Memory Cell?

- It's 8 bits long, so it can take 2^8 values
- The value range is thus $[0;255]$ (or $[-127;128]$ if signed)

How to store bigger values?

- For that, we aggregate memory cells, *i.e.* we interpret together adjacent cells
- `int` are stored on 4 cells Resulting range: $[0;2^{8\times4}[= [0;2^{32}[\approx [0;4e^{10}]$
- `short` are stored on 2 cells Resulting range: $[0;2^{16}[= [0;65\,535]$

Problem

- Impossible to interpret a memory area without infos on data type stored
- Remember: C memory is a big magma (never forget!)
- Veeery different from Java where you have introspection abilities

Chapter 3

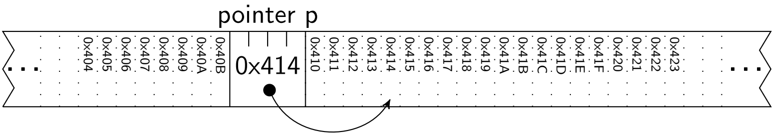
Memory Management in C

- Static Memory
  - Variables in C
  - Processes Memory Layout
  - Addresses
- Pointers
  - Basics
  - Pointers vs. Arrays
  - Casting Pointers
- Dynamic Memory
  - Memory Blocs and Pointers

Pointers

What is it?

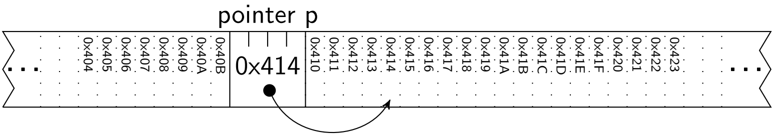
- Variable storing a memory address: Pointer value = rank of a memory cell
- On 32 bits, I need 4 bytes to store an address since biggest address$=2^{32\times 8}$ (8 bytes on 64 bits)
- Pointers are often written in hexadecimal (just a convention)
- Most of the time, numerical value is meaningless; where it points to is crucial



Pointers

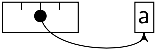
What is it?

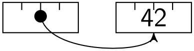
- Variable storing a memory address: Pointer value = rank of a memory cell
- On 32 bits, I need 4 bytes to store an address since biggest address=$2^{32\times 8}$ (8 bytes on 64 bits)
- Pointers are often written in hexadecimal (just a convention)
- Most of the time, numerical value is meaningless; where it points to is crucial



But we can't interpret memory areas w/o info on stored type!

- This information is given by the type of pointer

`char* pc;` 

`int* pi;` 

- It is possible to store the address of a pointer of a pointer: `int ***p;`

  Remember: types are to be read from right to left

Pointers Pitfalls

There is reasons why students don't like pointers

Pitfall #1: * has a very heavy semantic

- This little char is very loaded of semantic in C
- Forget only one * somewhere, and you're running into the segfault
  Same thing when writing a * too much

Pitfall #2: * actually has two differing meanings

- `int *p` declares a **pointer variable** `p` which is a pointer to an integer value
- `*p` is then the **pointed value**, interpreted according to the pointer type
- (that's actually three meanings when counting $\times$, the multiplication)

- `int *p; p=12;` selects where it points in memory
- `int *p; *p=12;` changes the memory in the pointed area

- Pascal was a bit more reasonable: `INTEGER ^p` vs. `p^` (at least other order)
- In Java, there is no pointers, but reference to objects are close to that concept

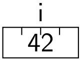
Retrieving the address of something

Motivation

- Knowing that your pointer `p` points to 0x2342 is almost never relevant
- Knowing that it points to your variable `i` is what you need

This is what the & operator does

- `int i=42;`

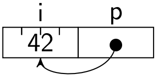


Retrieving the address of something

Motivation

- Knowing that your pointer `p` points to 0x2342 is almost never relevant
- Knowing that it points to your variable `i` is what you need

This is what the & operator does

- `int i=42; int *p=&i;` 

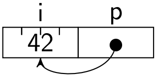
(successive variables are (often) adjacent)

Retrieving the address of something

Motivation

- Knowing that your pointer `p` points to 0x2342 is almost never relevant
- Knowing that it points to your variable `i` is what you need

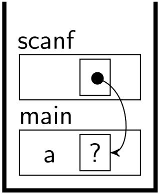
This is what the & operator does

- `int i=42; int *p=&i;` 

(successive variables are (often) adjacent)

We can now explain how scanf "modifies its arguments"

```
int main() {
  int a;
  scanf("%d",&a);
}
```



Stack

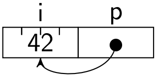
- `scanf` parameter: an address
  "%d" tells how to interpret it
- That's copied over, but that's fine
- `scanf` can modify the a variable,
  even if it's not in its scope
  (remember: C memory is a magma)
- other mystery:

Retrieving the address of something

Motivation

- Knowing that your pointer `p` points to 0x2342 is almost never relevant
- Knowing that it points to your variable `i` is what you need

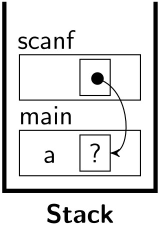
This is what the & operator does

- `int i=42; int *p=&i;` 

(successive variables are (often) adjacent)

We can now explain how scanf "modifies its arguments"

```
int main() {
  int a;
  scanf("%d",&a);
}
```

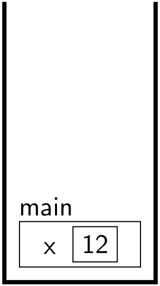


- `scanf` parameter: an address
  "%d" tells how to interpret it
- That's copied over, but that's fine
- `scanf` can modify the a variable,
  even if it's not in its scope
  (remember: C memory is a magma)
- other mystery: variable amount of params
  `man stdarg` ;)

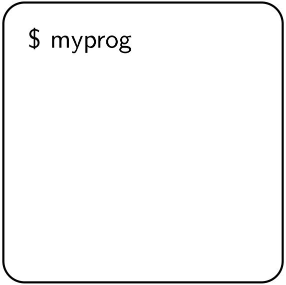
Fixing the triple() function

- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

```c
void triple(int *a) {
  *a=(*a)*3;
  return;
}

int main() {
  int x=12;
  triple(&x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

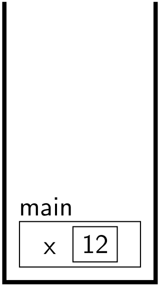


Output

Fixing the triple() function

- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

```c
void triple(int *a) {
  *a=(*a)*3;
  return;
}

int main() {
  int x=12;
  triple(&x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

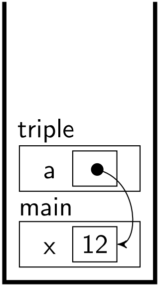
```
$ myprog
```

Output

Fixing the triple() function

- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

```c
void triple(int *a) {
  *a=(*a)*3;
  return;
}

int main() {
  int x=12;
  triple(&x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

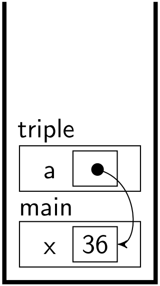
```
$ myprog
```

Output

Fixing the triple() function

- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

```
void triple(int *a) {
  *a=(*a)*3;
  return;
}

int main() {
  int x=12;
  triple(&x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

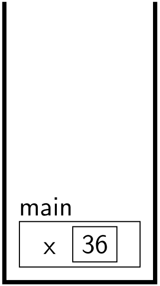
```
$ myprog
```

Output

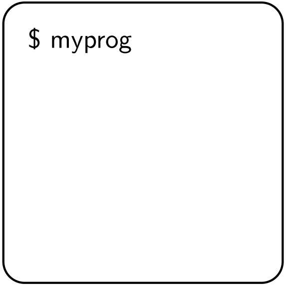
Fixing the triple() function

- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

```
void triple(int *a) {
  *a=(*a)*3;
  return;
}

int main() {
  int x=12;
  triple(&x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

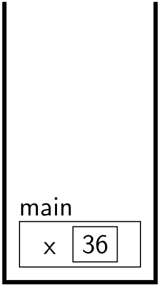


Output

Fixing the triple() function

- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

```c
void triple(int *a) {
  *a=(*a)*3;
  return;
}

int main() {
  int x=12;
  triple(&x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

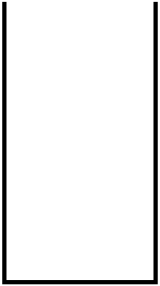
```
$ myprog
x: 36
```

Output

Fixing the triple() function

- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

```c
void triple(int *a) {
  *a=(*a)*3;
  return;
}

int main() {
  int x=12;
  triple(&x);
  printf("x: %d",x);
  return EXIT_SUCCESS;
}
```



Stack

```
$ myprog
x: 36
$
```

Output

- Pointers are powerful tools (that's why they are dangerous)

Pointers vs. Arrays

In C, Arrays are Pointers (at least, most of the time)

- Unfortunate heritage of C first years; One of the major pitfall for newcomers
- `char name[32];` pointer to a **reserved** area of 32 bytes
- `int ai[ ] = {0,1,2};` pointer to a reserved and inited area of 3 ints
- `void max(int ai[ ])` ≈ `void max(int *ai)` Expects an int pointer
- `void max(int ai[32])` Similar, but whole array is copied on stack
- When using `name` after `char name[32]` as if it were an automatic & `name`, when looked at as pointer, is the address of the first array cell
- This explains why strings don't take any & in scanf: they already are pointers

Considering Pointers as Arrays

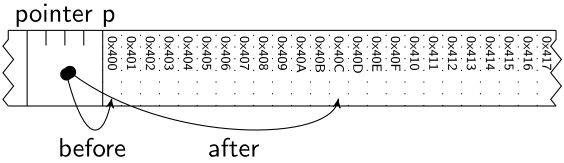
- `int *pi=...; pi[3];` This is valid; Behave as expected
  (no bound checking, as usual in C)

Pointer Arithmetic

Adding and subtracting integers to pointers is valid

- It represents a **shift in cell (not in bytes)**

```
int *pi=0x400;
pi=pi+3;
printf("pi:%x\n",pi);
```



- Value change in *pi: `value_after`$=$ `value_before`$+$`sizeof(int)`$\times 3$ because it points on integers

Subtracting 2 pointers is valid

- It gives the shift between them (in cells, not in byte)

Other arithmetic operations are **not valid** on pointers

Pointers, Arithmetic, and Arrays

- `p[i]` is equivalent to `*(p+i)` (yes, C notations about arrays are messy)

Chapter 3

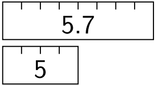
Memory Management in C

Casting Data

What is it?

- This is the well known $\boxed{\texttt{int a = (int)b}}$ notation. More generally, $\boxed{\texttt{(type)}}$
- It is used to convert something in a type into something else
- Two meanings, depending on whether it's applied on scalars or pointers
- Quite the same story in Java, actually

Casting Scalars: Converting values

- ```
  double d = 5.7;
  int i = (int)d;
  ```
  ```

Casting Data

What is it?

- This is the well known $\boxed{\texttt{int a = }\textbf{(int)}\texttt{b}}$ notation. More generally, $\boxed{\texttt{(type)}}$
- It is used to convert something in a type into something else
- Two meanings, depending on whether it's applied on scalars or pointers
- Quite the same story in Java, actually

Casting Scalars: Converting values

- ```
  double d = 5.7;
  int i = (int)d;
  ```



- Casting Scalars can lead to:
  - Change the memory representation of the value
  - Change the amount of memory needed to represent the value
  - Lead to precision loss (!)
  ```

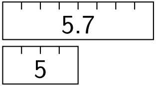
Casting Data

What is it?

- This is the well known `int a = (int)b` notation. More generally, `(type)`
- It is used to convert something in a type into something else
- Two meanings, depending on whether it's applied on scalars or pointers
- Quite the same story in Java, actually

Casting Scalars: Converting values

```
double d = 5.7;
int i = (int)d;
```



- Casting Scalars can lead to:
  - Change the memory representation of the value
  - Change the amount of memory needed to represent the value
  - Lead to precision loss (!)

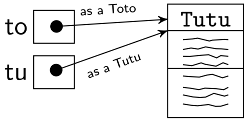
Casting Pointers: Changing the semantic

- It's written exactly the same way ... but the meaning is very different
- Let's look again at the Java semantic of reference casting

Casting Objects in Java

Java Semantic Casting

```
Toto to = new Tutu();
Tutu tu = (Tutu)to;
```



- Through `tu`, I consider the object to be a `Tutu`
- It does not change the value of the object, only what I expect from it
- Only valid if `Tutu extends Toto` (and useless if `Toto extends Tutu`)

Side note: Static vs. Dynamic typing is a creepy part of Java

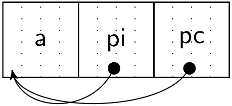
- Casts relax constraints at compilation time; Enforced at execution time
  That is what `TypeCastException` is made for
- Guessing which method gets called is sometimes excessively difficult
  Check again TD4 of POO if you forgot
- But it's hard to depreciate the Java typing system in a course on C...

Casting Pointers in C

They change the Pointer Semantic

- The numeric value of the pointer does not change
- But the dereferencing it completely different
- Also has a huge impact on pointer arithmetic

```
int a;
int *pi=&a;
char *pc=pi;
pi++;
pc++;
```

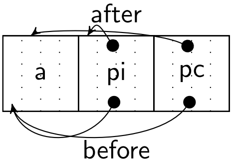


Casting Pointers in C

They change the Pointer Semantic

- The numeric value of the pointer does not change
- But the dereferencing it completely different
- Also has a huge impact on pointer arithmetic

```
int a;
int *pi=&a;
char *pc=pi;
pi++;
pc++;
```



Generic Pointers

Generic pointers are sometimes handy

- To describe pointers that can point to differing data
  Example: in scanf, how to interpret the pointer is given by the format
- To describe pointers to *raw* data (ie, you don't care about the pointed type)
  Example: When copying memory chunk over, content does not matter

That is what void* is made for

- Modern compiler even allow you to do pointer arithmetic on them
  supposing that `sizeof(void)=1`, which is ... arbitrary
- Older compiler force you to cast them to `char*` before

Chapter 3

Memory Management in C

Dynamic Memory

Motivation

- Arrays are statically sized in C (*i.e.* their size must be known at compilation)
- It is forbidden to write:

```
int n;
scanf("%d",&n);
int tab[n];
```

  because `n` is only known at execution
- (this is not true in C99, but C99 not widely spread yet)

Solution

- Directly request memory chunks from the system
- Manage them yourself
- And return them to the system when you're done

Remember the Memory Layout of a Process

Data	**Heap**		**Stack**
Code+Globals	Dynamic Memory	$\rightarrow \emptyset \leftarrow$	Stack Frames

0 4Gb

- The idea is to request memory from the heap

Requesting Memory Chunks from the heap

The several ways of doing so

- As usual, there is a high level and a low level API
- At low-level, the `brk()` syscall allows to move the heap boundary
  And you are on your own to manage its content (emacs does it)

malloc() and friends

- This higher level API directly gives memory chunks in heap
  and deal automatically with `brk()`
- There is only 3 functions to know

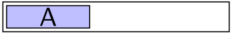
`#include <stdlib.h>`	
`void*malloc(int size)`	Request a new memory chunk
`void free(void*p)`	Return a memory chunk
`void*realloc(void*p,int size)`	Expend a memory chunk

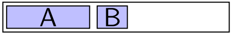
Understanding malloc and friends

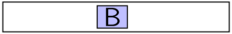
Function Semantic

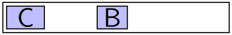
- malloc() requests a new memory chunk and return the address of beginning
  If there is not enough free memory, it returns NULL

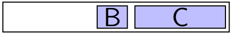
Think of a land registry for the memory

- `void *A=malloc(12);` 

- `void *B=malloc(5);` 

- `free(A);` 

- `void *C=malloc(6);` 

- `C=realloc(C,13);` 

As usual in C

- There is no protection mechanism here: Mess it up and you'll get a segfault
- Two surviving strategies:
  - Avoid issues through best practices
  - Solve issues through debugging tools

Best Practices about Dynamic Memory

Rule #1: Only access to reserved areas

- Land Registry Analogy: Only build stuff on land that you own

```
int *A;
*A=1;
A=malloc(sizeof(int));
```

Error! A used before malloc!

(buy it before building)

```
int *A=malloc(sizeof(int));
free(1);
*A=1;
```

Error! A used after free!

(forget it after selling it)

- You'll have similar symptoms in both case
  - If you are lucky, segfault (error signaled where the fault is)
  - If not, some memory pollution (probably a later segfault, harder to diagnose)

Best Practices about Dynamic Memory

Rule #2: To any malloc(), one and only one free()

- If you forget the `free()`, there is a **memory leak**
  - The system assumes that this area is used where it's not anymore
  - Ok to have a few memleaks. Too much of them will exhaust system resources
  - Slows everything down (swapping), and malloc() will eventually return NULL
- If you call `free()` twice (**double free**), strange things will occur

```
int *A=malloc(12);
free(A);
int *B=malloc(12);
free(A);
```

$\rightsquigarrow$ Probably frees B ...
Unfriendly if A and B are in two separate modules

  - That is why modern malloc implementations try to detect this situation
  - And kill faulty program as soon as the error is detected