Mastering your Linux: C and Shell Programming

Martin Quinson <martin.quinson@loria.fr>

Telecom Nancy – 1ère année

2014-2015

Introduction

Course Goals

- Help you mastering your thid programming language
  - Basics about the syntax
  - Caveats (of memory management, amongst other)
  - Get some good style
- Help you mastering your Linux box (or any other UNIX-based one)
  - Fluent use of the terminal
  - Non-trivial command lines
  - Simple scripts

Prerequisite

- Algorithmic Background: you cannot program without that
- Scala/Java Programming: we won't learn to program, but how to write it in C

Course Context at Telecom Nancy

- Part of Programing Track (courses PPP, TOP, POO, SD, CSH)
- Starts a new track on Operating System (courses CSH, RS, RSA)

Administrativae

Module Time Table

- Three lectures
- 7 practical labs + 3 repetition sessions (+ exam): The C language
- 6 practical labs + 2 small group lectures (+ exam): Shell Scripting

Evaluation

- Test on table *(partiel)* on C language
  - What: Content of lectures and labs (of course)
  - When: someday in march (check ADE agenda)
  - Allowed material during test: one A4 sheet of paper only
    - Hand-written (not typed)
    - From you (no photocopy)
- Homework: Do whatever you want (in C)
- Test on table on Shell Scripting
  - When: someday in may (check ADE agenda)
  - (Ask Suzanne Collin for details)

About me

Martin Quinson

- Study: Université de Saint Étienne, France
- PhD: Grids and HPC in 2003 (team Graal of INRIA / ENS-Lyon, France)
- Since 2005:
  - Assistant professor at Telecom Nancy (Université de Lorraine)
  - Researcher of Algorille team (soon of Veridis) of LORIA/Inria
- Research interests:
  - Context: Distributed Systems (Grids, HPC, Clusters)
  - Main: Simulation of Distributed Applications (SimGrid project)
  - Others: Experimental Methodology, Model-Checking, ...
- Teaching duties:

  1A: PPP: introduction to Java; TOP: Technics and tOols for Programming; CSH: C as Second Language (and Shell)
  2A: RS: System Programming (and Networking)
  3A: Peer-to-Peer Systems and Advanced Distributed Algorithms (master)
- More infos:
  - `http://www.loria.fr/~quinson` (`Martin.Quinson@loria.fr`)

References: Courses on Internet

- Introduction to Systems Programming (C. Grothoff)
  C covered, but not only.
  `http://grothoff.org/christian/teaching/2009/2355/`
- C / Shell (A. Crouzil, J.D. Durou et Ph. Joly; U. Paul Sabatier, Toulouse)
  Good coverage of the whole module (in French).
  `http://www.irit.fr/ACTIVITES/EQ_TCI/ENSEIGNEMENT/CetSHELL/`
- Support de Cours de Langage C (Christian Bac; Telecom SudParis)
  The C Language (in French).
  `http://picolibre.int-evry.fr/projects/svn/coursc/`

Table of Contents

- Introduction and Generalities
  - Introduction; Motivation; History.

1. Part I: C as Second Language
   - Syntax and Basic usage
     - Introduction; First C program and compilation; Syntax, printf; C *vs.* Java.
   - Memory Management in C
     - Variable visibility, storage class; Malloc and friends; Debugging problems.
   - Advanced C Topics
     - Modularity in C; Makefile; Performance tuning; Game programming.

2. Part II: Shell Scripting
   - Low Script-fu knowledge
     - Introduction; First shell "scripts"; Redirecting I/O & Pipes; basic commands.
   - Medium Script-fu knowledge
     - More Syntax for Advanced Scripts; Not so basic commands.
   - Advanced Script-fu knowledge
     - Shell functions; Variable Substitutions; Sub-shells; Arrays.

Chapter 1

C and Unix

- Introduction
  - C? UNIX? What is all this about?
  - Why do we need to study C?
  - Why do we need to study C and UNIX together?
- C as Second Language
  - C vs. Java
  - How to survive in C?
  - Your first C program
- First steps in Unix
  - Désignation des fichiers
  - Protection des fichiers
  - Using the terminal

C? UNIX? What is all this about?

Let's go for a little pool, please

- Who never heard the word "Unix" *before arriving* at Telecom Nancy?

C? UNIX? What is all this about?

Let's go for a little pool, please

- Who never heard the word "Unix" *before arriving* at Telecom Nancy?
- Who in the room have Linux installed on a computer at home?

C? UNIX? What is all this about?

Let's go for a little pool, please

- Who never heard the word "Unix" *before arriving* at Telecom Nancy?
- Who in the room have Linux installed on a computer at home?
- Who have a network of Linux boxes at home?

C? UNIX? What is all this about?

Let's go for a little pool, please

- Who never heard the word "Unix" *before arriving* at Telecom Nancy?
- Who in the room have Linux installed on a computer at home?
- Who have a network of Linux boxes at home?

Telecom Nancy population very heterogeneous

- Usually about $\frac{1}{3}$ didn't heard about Unix before arriving, and $\frac{1}{3}$ use it already
- We are here to level everybody
- Yep, some of you already know the first lectures (go get some maths)
- But be patient, soon, everyone will be lost (including YOU!)

C? UNIX? What is all this about?

Let's go for a little pool, please

- Who never heard the word "Unix" *before arriving* at Telecom Nancy?
- Who in the room have Linux installed on a computer at home?
- Who have a network of Linux boxes at home?

Telecom Nancy population very heterogeneous

- Usually about $\frac{1}{3}$ didn't heard about Unix before arriving, and $\frac{1}{3}$ use it already
- We are here to level everybody
- Yep, some of you already know the first lectures (go get some maths)
- But be patient, soon, everyone will be lost (including YOU!)

Further Quizz

- Could you define Unix in a word?

C? UNIX? What is all this about?

Let's go for a little pool, please

- Who never heard the word "Unix" *before arriving* at Telecom Nancy?
- Who in the room have Linux installed on a computer at home?
- Who have a network of Linux boxes at home?

Telecom Nancy population very heterogeneous

- Usually about $\frac{1}{3}$ didn't heard about Unix before arriving, and $\frac{1}{3}$ use it already
- We are here to level everybody
- Yep, some of you already know the first lectures (go get some maths)
- But be patient, soon, everyone will be lost (including YOU!)

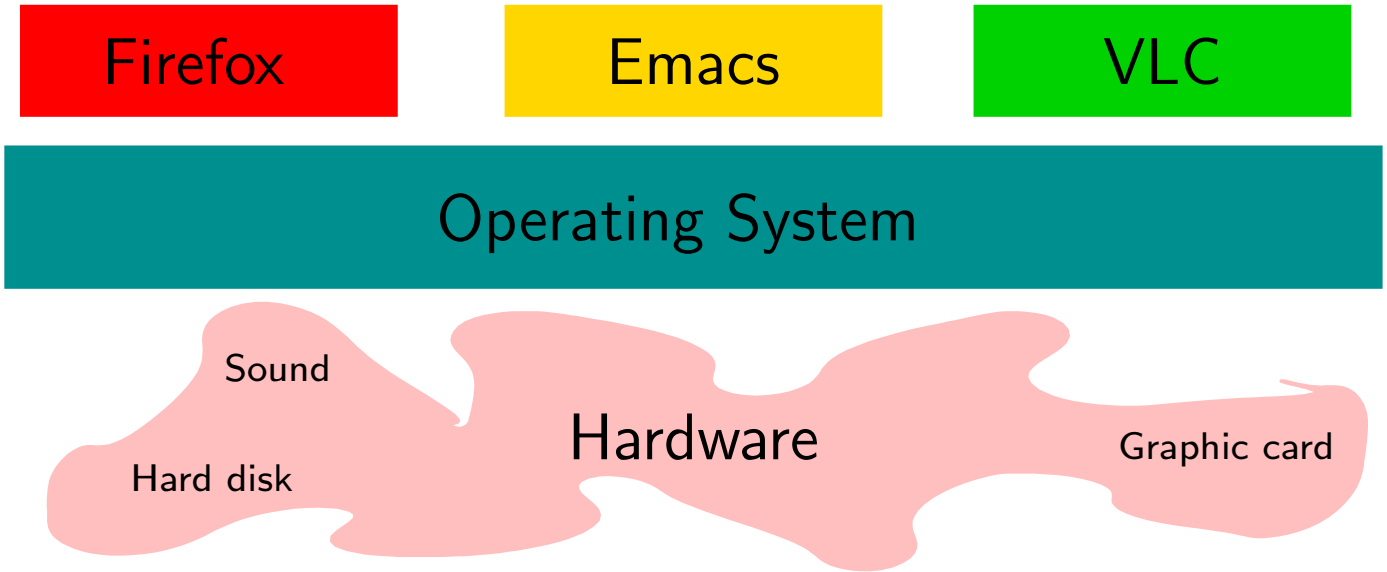
Further Quizz

- Could you define Unix in a word? That's an Operating System

Operating System

What is an Operating System?

- That's the software between the applications and the hardware
- Handles (and protects) the resources
- Offers an unified interface to the applications



Operating System Basics

- What's the Operating System on neptune host (where you do your labs)?

Operating System Basics

- What's the Operating System on neptune host (where you do your labs)?

  That's Linux
- What's the difference between that and the Unix we spoke about before?

Operating System Basics

- What's the Operating System on neptune host (where you do your labs)?

  That's Linux
- What's the difference between that and the Unix we spoke about before?

  $Linux \in Unix$, ie Linux is one member of the greater Unix family
- What other Operating System you know?

Operating System Basics

- What's the Operating System on neptune host (where you do your labs)?

  That's Linux
- What's the difference between that and the Unix we spoke about before?

  $Linux \in Unix$, ie Linux is one member of the greater Unix family
- What other Operating System you know?

  Windows, Mac OS are the most popular
- Any idea of the amount of existing Operating System? Guess the count

Operating System Basics

- What's the Operating System on neptune host (where you do your labs)?

  That's Linux
- What's the difference between that and the Unix we spoke about before?

  $Linux \in Unix$, ie Linux is one member of the greater Unix family
- What other Operating System you know?

  Windows, Mac OS are the most popular
- Any idea of the amount of existing Operating System? Guess the count

  Writing an OS is a classical practical lab, so there's really a lot of them
- What's the link between Mac OS and the other OSes?

Operating System Basics

- What's the Operating System on neptune host (where you do your labs)?

  That's Linux
- What's the difference between that and the Unix we spoke about before?

  $Linux \in Unix$, ie Linux is one member of the greater Unix family
- What other Operating System you know?

  Windows, Mac OS are the most popular
- Any idea of the amount of existing Operating System? Guess the count

  Writing an OS is a classical practical lab, so there's really a lot of them
- What's the link between Mac OS and the other OSes?

  Mac OS 9 was a specific OS family (but it's long dead); Mac OS X is a Unix
- Why don't we speak of Windows instead?

Operating System Basics

- What's the Operating System on neptune host (where you do your labs)?
  That's Linux
- What's the difference between that and the Unix we spoke about before?
  $Linux \in Unix$, ie Linux is one member of the greater Unix family
- What other Operating System you know?
  Windows, Mac OS are the most popular
- Any idea of the amount of existing Operating System? Guess the count
  Writing an OS is a classical practical lab, so there's really a lot of them
- What's the link between Mac OS and the other OSes?
  Mac OS 9 was a specific OS family (but it's long dead); Mac OS X is a Unix
- Why don't we speak of Windows instead?
  (not *'cause it sucks*) Because Windows is quite too different from Unix
- If so, why do we speak of Unix anyway?
  1.
  2.

Operating System Basics

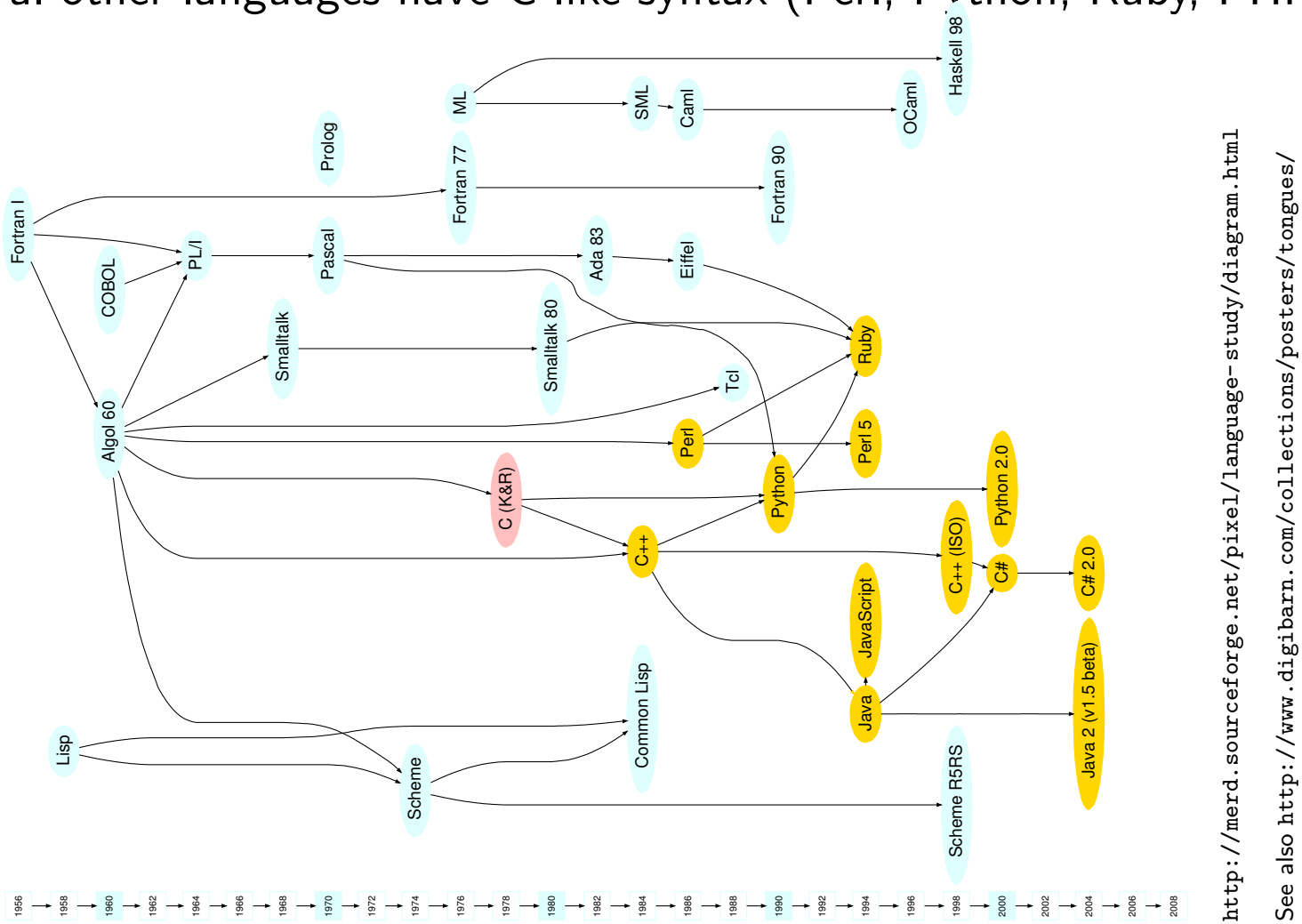
- What's the Operating System on neptune host (where you do your labs)?
  That's Linux
- What's the difference between that and the Unix we spoke about before?
  $Linux \in Unix$, ie Linux is one member of the greater Unix family
- What other Operating System you know?
  Windows, Mac OS are the most popular
- Any idea of the amount of existing Operating System? Guess the count
  Writing an OS is a classical practical lab, so there's really a lot of them
- What's the link between Mac OS and the other OSes?
  Mac OS 9 was a specific OS family (but it's long dead); Mac OS X is a Unix
- Why don't we speak of Windows instead?
  (not *'cause it sucks*) Because Windows is quite too different from Unix
- If so, why do we speak of Unix anyway?
  1. It's the most widespread OS (if you include servers, embedded, etc ;)
  2.

Operating System Basics

- What's the Operating System on neptune host (where you do your labs)?
  That's Linux
- What's the difference between that and the Unix we spoke about before?
  $Linux \in Unix$, ie Linux is one member of the greater Unix family
- What other Operating System you know?
  Windows, Mac OS are the most popular
- Any idea of the amount of existing Operating System? Guess the count
  Writing an OS is a classical practical lab, so there's really a lot of them
- What's the link between Mac OS and the other OSes?
  Mac OS 9 was a specific OS family (but it's long dead); Mac OS X is a Unix
- Why don't we speak of Windows instead?
  (not *'cause it sucks*) Because Windows is quite too different from Unix
- If so, why do we speak of Unix anyway?
  1. It's the most widespread OS (if you include servers, embedded, etc ;)
  2. Because it's heavily connected to C (Why should we study the C language?)

Why should we study the C language? Huge impact

- C++ is an object extension of C (you cannot master C++ without C)
- Java is some sort of (safe) subset of C++; C# is a variation of Java
- Several other languages have C-like syntax (Perl, Python, Ruby, PHP)

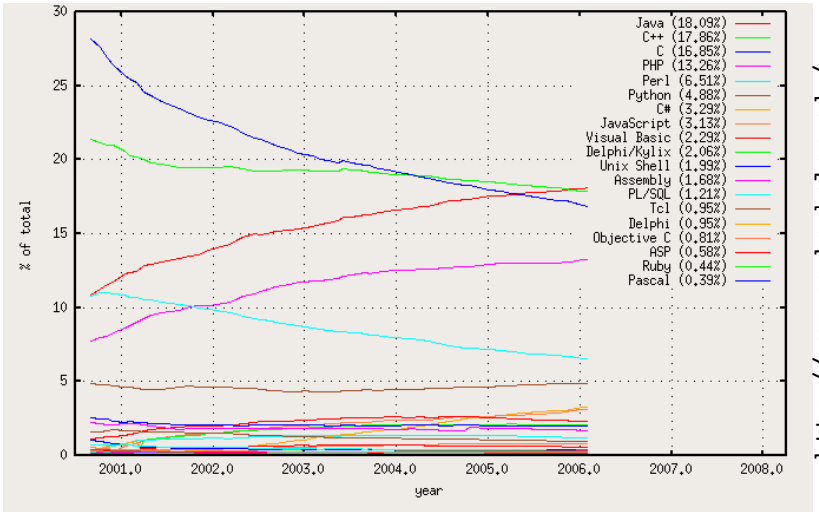


http://merd.sourceforge.net/pixel/language-study/diagram.html

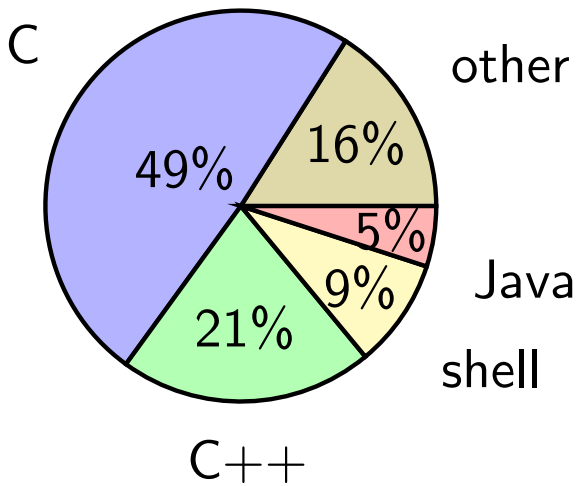
See also http://www.digibarn.com/collections/posters/tongues/

Why should we study the C language? Widespread

- De facto standard for System Programming: Windows, OS X, Linux, BSD in C
- Counting SourceForge projects. Java: 18%; C++: 17.9%; C: 15.9%



- Counting SLOC in Debian. Quite different numbers...



More details:
`http://debian-counting.libresoft.es/lenny/`

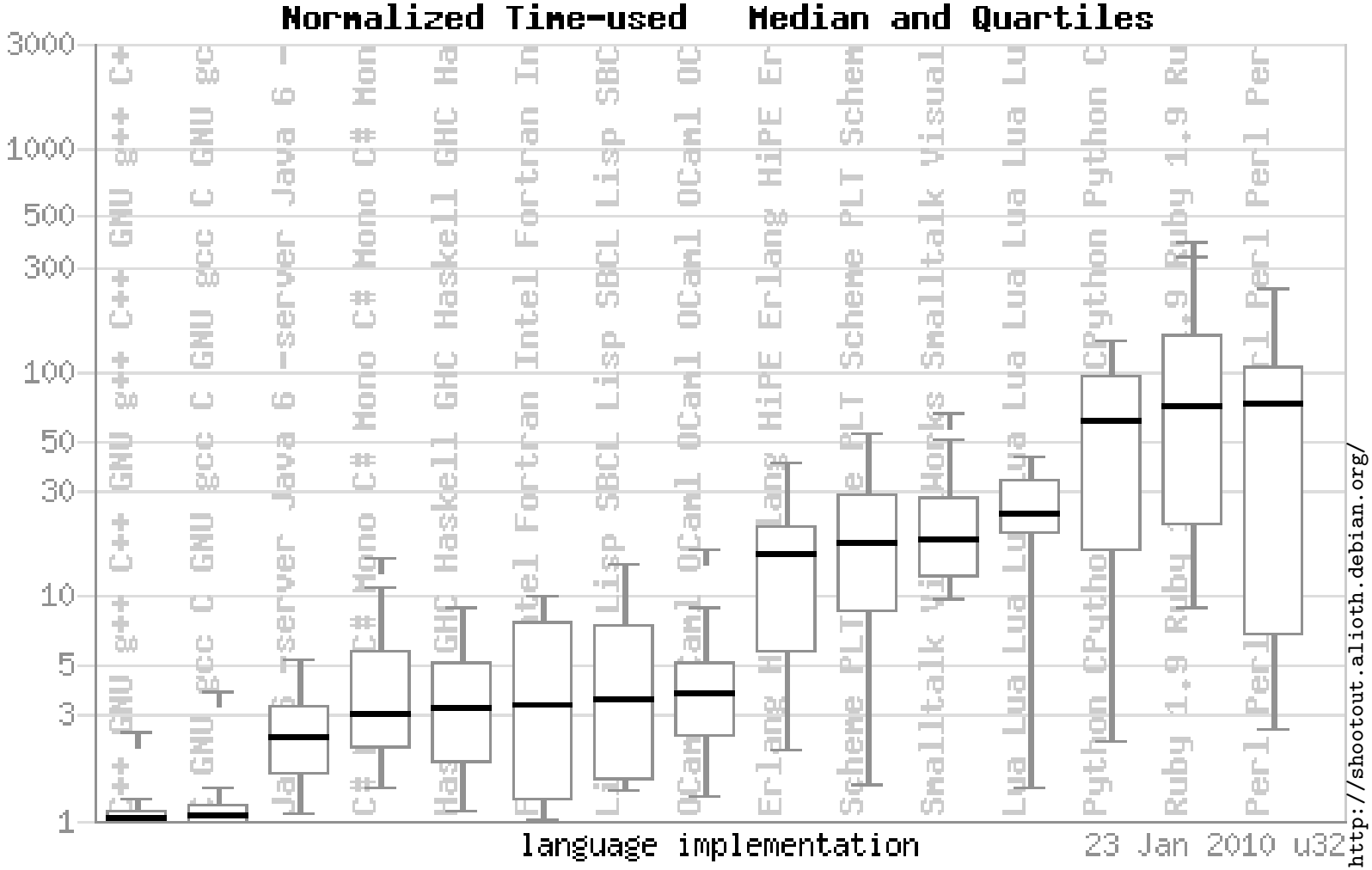
See also: `http://www.dwheeler.com/sloc/`

- Big codes are in C/C++
- Toy projects tend to be in Java

(but things change)

Why should we study the C language? Fast

- C program typically execute faster than in other programs



- One could argue that this is because it has the best tools, but not only

Studying the C language for educational purpose

Understanding C helps you understanding the system as a whole

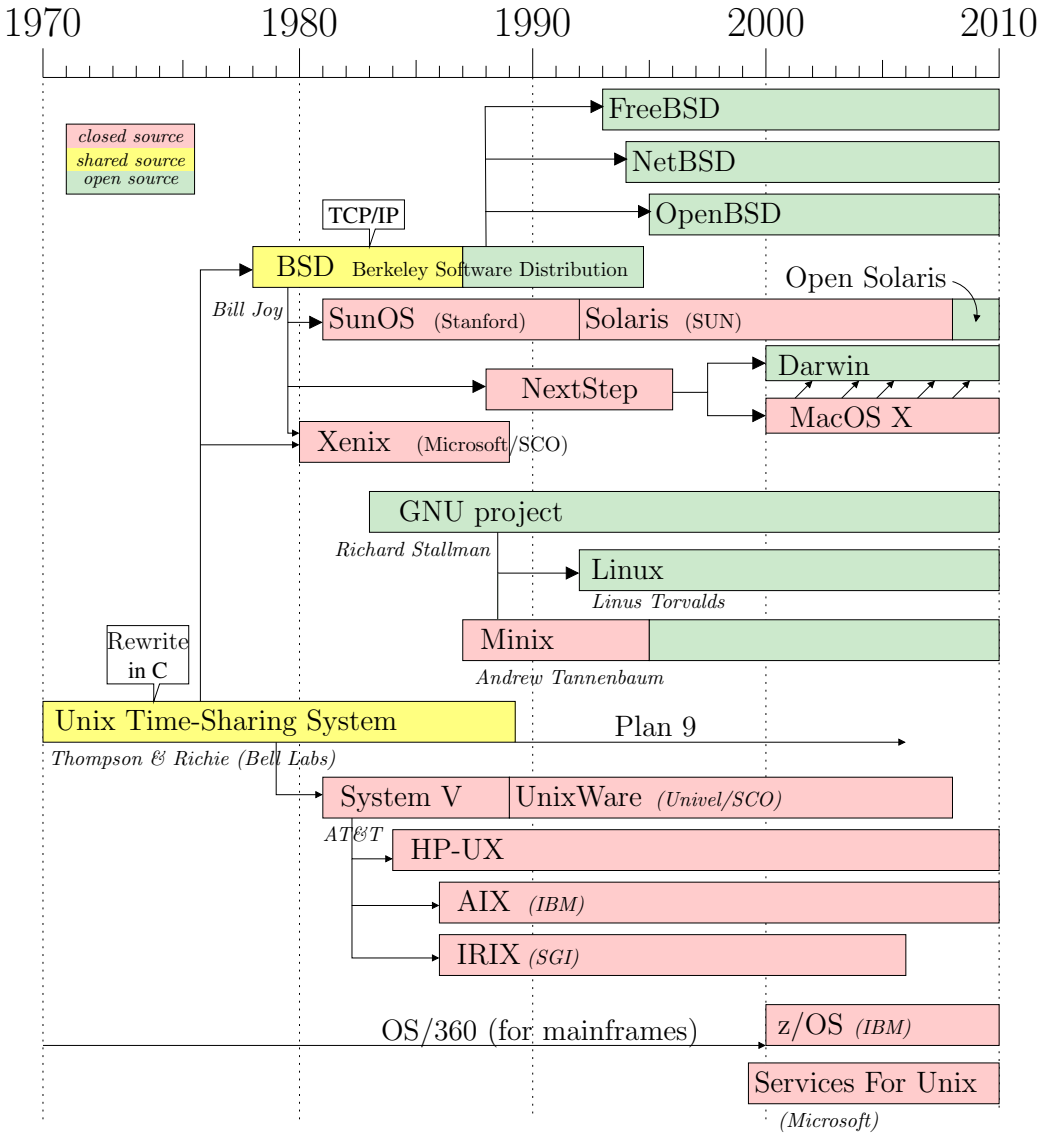
- C is the closest high-level language to the machine
- Every OS are written in C, so lower interface is in C/C++ too
- OS/hardware co-evolution: C conceptual model describes most hardware

Understanding C helps you writing better Java code

- Java/.Net/Perl/etc hide underlying low-level mechanism
- But these mechanism can be very important (to performance for example)
- To understand how objects get passed by ref, realize that they are pointers

Why do we need to study C and UNIX together?

Because they were invented together!



Unix history

1965 MULTICS: ambitious system project (Bell Labs)
1969 Bell Labs give up MULTICS, UNICS begins
1970 Unix: Official Bell Labs project
1973 Rewrite in C
Distribution to universities
Sold by AT&T
80-90 Unix War: BSD vs. System V
90-10 Normalization Effort (POSIX)

C history

1967 BCPL used at Bell Labs;
1968 B [Thompson]: simplification
1971 C K&R (somehow typed)
1983 C++: object oriented
1989 ANSI C; 1990: ISO C (C90)
1999 ISO C updated (C99)

What is Special About C?

Low Level: sort of abstract assembly language of historical processors

- Was invented on a PDP-11 with 24kb of memory: KISS!
- Process memory visible as an array of bytes
- Nothing in the language will prevent you from doing (really) stupid things

C combines the power of assembler with the portability of assembler.

Extensible: most higher-level features doable in C

- Self-modifying code, Introspection, Code migration, etc. (but all by yourself)
- (actually, JVM partially written in C/C++)

If you can't do it in ANSI C, it isn't worth doing.

Relatively Stable: almost backward compatible since seventies

- Other languages got heavily lifted too often (but some heritages unpleasant)

C has hardly any runtime system

- Small footprint, easily ported to new architectures (need to reinvent the wheel)

So, should we use C once studied?

Benefits

- More control over the execution behaviour of programs
- More opportunity for performance tuning
- Access to low-level features of system

Disadvantages

- Need to do your own memory management
- Typically takes more lines of code to accomplish each task
- More opportunity to make mistakes

Summary

- C is a powerful programming tool for experts
- Presents many potential hazards for novices
- Helps you to understand low-level execution ideas
- Helps transforming you from a novice to an expert

$\rightsquigarrow$ Use it when you need it, avoid it when you don't need it

Chapter 1

C and Unix

- Introduction
  - C? UNIX? What is all this about?
  - Why do we need to study C?
  - Why do we need to study C and UNIX together?
- C as Second Language
  - C vs. Java
  - How to survive in C?
  - Your first C program
- First steps in Unix
  - Désignation des fichiers
  - Protection des fichiers
  - Using the terminal

C as Second Language

Similarities between C and Java

- Operators
  - Arithmetic (+,-,*,/,% ; ++,- -,*=,...); Bitwise (&,|,^,!,<<,>>)
  - Relational (<,>,<=,>=,==,!=); Logical &&, ||, !, (a?b:c)
- Keywords and Language Constructs
  - `if( ){ } else { }`
  - `while( ){ }`
  - `switch( ) { case 0: ... }`
  - `for(i=0; i<100; i++){ }`
  - `do { } while( )`
  - `break`, `continue`, `return`
- Basic (primitive) types: void, int, short, long; float, double; char.
  No boolean, use int instead (0=False; anything else=True)
- Function declarations: `int fact(int a){return a==0 ? 1 : a*fact(a-1);}`

Differences between C and Java

- No exception: usually rely on int error code instead (and usually a pain)
- No class/package/interface: code modularity different (not compiler-enforced)
- No garbage collector: alloc and free manually needed memory (incredible pain)

C as Second Language

C seems familiar when you know Java

- Actually, that's Java which is highly inspired from C/C++
- Feels like a Java without any object but with full access to everything

C is like Java without comfort and without any protections

- Standard library is poor (but huge amount of extensions)
- Compiler is incredibly permissive (by default)
- It's possible to shoot yourself in the foot in Java, that's common in C
- On error, Java displays a stack trace, C spits "segfault" or "invalid free" errors

> *Unix was not designed to stop people from doing stupid things, because that would also stop them from doing clever things.*
>
> – Doug Gwyn

C main specificities in a Nutshell

- Memory fully accessible through *pointers*
- Arrays are handled as pointer to memory
- Declaration syntax very similar to usage syntax (to the price of readability)

How to survive in C?

Use the tools that can help you

- Use the compiler warning flags `-Wall` mandatory, other usefull
- Use a proper editor (able of colorization, auto-indent, compile easily)
  - Good editors: emacs & vi (historical), Eclipse/CDT (my personal favorite)
  - Bad editor: gedit (not good for text, BAD for code)
- The debugger (`gdb`) must become your friend quickly
- valgrind is a piece of magic (C coding without it is masochism)

Don't assume you're a genius (ie, don't do stupid things — yet)

- Pay attention to the modularity of your code (not compiler-enforced anyhow)
- Document your code (with readable comments, or doxygen for bigger projects)
- Get some discipline (coding convention), and stick to it
  - Symbol naming (my_variable or myVariable), indentation, etc.
  - Which one is not very important. Pick one, and stick to it
- Keep it simple: it's easy to write unreadable C code

Bad Style Coding as a Game

The International Obfuscated C Code Contest (www.ioccc.org)

- Yearly contest of intentionally obfuscated codes (in C; exist for other languages)

Example: (arachnid, 2004 entry)

```
#include <ncurses.h>/*****************************************************/
            int                 m[256                     ] [         256   ],a
 ,b   ;;;   ;;;   WINDOW*w;    char*l=""    "\176qxl"    "q"    "q"    "k"    "w\
xm"    "x"    "t"          "j"          "v"          "u"          "n"          ,Q[
 ]=    "Z"    "pt!ftd‘"    "qdc!‘eu"    "dq!$c!nnwf"/**    ***    */"t\040\t";c(
int    u ,          int          v){                       v?m    [u]          [v-
 1]    |=2,m[u][v-1] &    48?W][v-1    ] &    15]]):0:0;u?m[u    -1][v]|=1    ,m[
 u-                1][    v]&          48?                W-1    ][v          ]&
15]    ]):0:0;v<    255    ?m[    u][v+1]|=8,m[u][v+1]&    48?    W][    v+1]&15]]
):0          :0;          u <                 255    ?m[    u+1          ][v    ]|=
4,m[u+1][    v]&48?W+1][v]&15]]):0:0;W][    v]&    15]    ]);}cu(char*q){    return
 *q                ?cu    (q+          1)&          1?q    [0]                ++:
q[0    ]--    :1;    }d(    int    u ,    int/**/v,    int/**/x,    int    y){    int
Y=y    -v,    X=x          -u;    int          S,s    ;Y<          0?Y    =-Y    ,s,
s=-    1:(    s=1);X<0?X=-X,S    =-1   :(S=    1);    Y<<=    1;X<<=1;    if(X>Y){
int    f=Y                -(X    >>1    );;                while(u!=          x){
f>=    0?v+=s,f-=X:0;u    +=S    ;f+=    Y;m[u][v]|=32;mvwaddch(w,v    ,u,    m[u
 ][                v]&    64?    60:          46)          ;if          (m[    u][
v]&16){c(u,v);;    ;;;    ;;;    return;}}    }else{int    f=X    -(Y>>1);;    while
 (v    !=y          ){f    >=0          ?u    +=S,                f-=          Y:0
 ;v    +=s    ;f+=X;m[u][v]|=    32;mvwaddch(w,v    ,u,m[u][v]&64?60:46);if(m[u
 ][                    v]&          16)    {c(    u,v                          );
  ;    return;;;}}}}Z(    int/**/a,    int    b){    }e(    int/**/y,int/**/    x){
int                i ;          for          (i=          a;i                <=a
+S;i++)d(y,x,i,b),d(y,x,i,b+L);for(i=b;i<=b+L;i++)d(y,x,a,i),d(y,x,a+    S,i
 );                    ;;;          ;;;          ;;;                ;;;    ;
  mvwaddch(w,x,y,64);    ;;;    ;;;    ;;;    prefresh(    w,b,a,0,0    ,L-    1,S-1
);}             main(          int                V ,    char                *C[
  ]    ){FILE*f=    fopen(V==1?"arachnid.c"/**/    :C[    1],"r");int/**/x,y,c,
                (source code cut here)
```

Bad Style Coding as a Game

The International Obfuscated C Code Contest (www.ioccc.org)

- Yearly contest of intentionally obfuscated codes (in C; exist for other languages)

Example: Full (interactive) Maze Escape Game (arachnid, 2004 entry)

```
#include <ncurses.h>/*****************************************************/
            int                m[256                   ] [         256   ],a
 ,b   ;;;   ;;;   WINDOW*w;   char*l=""   "\176qxl"   "q"   "q"   "k"   "w\
xm"   "x"   "t"         "j"         "v"         "u"         "n"         ,Q[
 ]=   "Z"   "pt!ftd`"   "qdc!`eu"   "dq!$c!nnwf"/**   ***   */"t\040\t";c(
int   u ,         int         v){                     v?m   [u]         [v-
 1]   |=2,m[u][v-1] &   48?W][v-1   ] &   15]]):0:0;u?m[u   -1][v]|=1   ,m[
 u-               1][   v]&         48?               W-1   ][v         ]&
15]   ]):0:0;v<   255   ?m[   u][v+1]|=8,m[u][v+1]&   48?   W][   v+1]&15]]
):0         :0;         u <               255   ?m[   u+1         ][v   ]|=
4,m[u+1][   v]&48?W+1][v]&15]]):0:0;W][   v]&   15]   ]);}cu(char*q){   return
 *q               ?cu   (q+         1)&         1?q   [0]               ++:
q[0   ]--   :1;   }d(   int   u ,   int/**/v,   int/**/x,   int   y){   int
Y=y   -v,   X=x         -u;   int         S,s   ;Y<         0?Y   =-Y   ,s,
s=-   1:(   s=1);X<0?X=-X,S   =-1  :(S=   1);   Y<<=   1;X<<=1;   if(X>Y){
int   f=Y               -(X   >>1   );;               while(u!=         x){
f>=   0?v+=s,f-=X:0;u   +=S   ;f+=   Y;m[u][v]|=32;mvwaddch(w,v   ,u,   m[u
 ][               v]&   64?   60:         46)         ;if         (m[   u][
v]&16){c(u,v);;   ;;;   ;;;   return;}}   }else{int   f=X   -(Y>>1);;   while
 (v   !=y         ){f   >=0         ?u   +=S,               f-=         Y:0
 ;v   +=s   ;f+=X;m[u][v]|=   32;mvwaddch(w,v   ,u,m[u][v]&64?60:46);if(m[u
 ][                     v]&         16)   {c(   u,v                     );
  ;   return;;;}}}}Z(   int/**/a,   int   b){   }e(   int/**/y,int/**/  x){
int               i ;         for         (i=         a;i               <=a
+S;i++)d(y,x,i,b),d(y,x,i,b+L);for(i=b;i<=b+L;i++)d(y,x,a,i),d(y,x,a+   S,i
 );                     ;;;         ;;;         ;;;               ;;;   ;
  mvwaddch(w,x,y,64);   ;;;   ;;;   ;;;   prefresh(   w,b,a,0,0   ,L-   1,S-1
);}             main(         int               V ,   char              *C[
  ]   ){FILE*f=   fopen(V==1?"arachnid.c"/**/   :C[   1],"r");int/**/x,y,c,
                (source code cut here)
```

Bad Style Coding as a Game

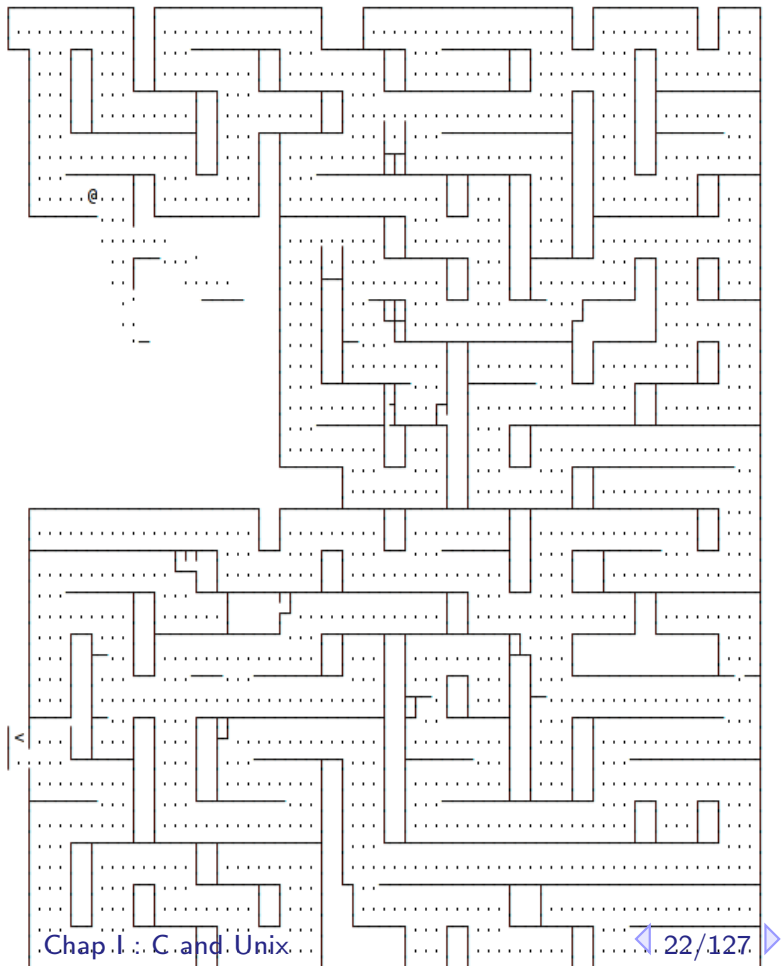
The International Obfuscated C Code Contest (www.ioccc.org)

- Yearly contest of intentionally obfuscated codes (in C; exist for other languages)

Example: Full (interactive) Maze Escape Game (arachnid, 2004 entry)

```
#include <ncurses.h>/*****************************************************/
            int                m[256                    ] [         256   ],a
 ,b   ;;;   ;;;   WINDOW*w;   char*l=""   "\176qxl"   "q"   "q"   "k"   "w\
xm"   "x"   "t"         "j"         "v"         "u"         "n"         ,Q[
 ]=   "Z"   "pt!ftd`"   "qdc!`eu"   "dq!$c!nnwf"/**   ***   */"t\040\t";c(
int   u ,         int         v){                      v?m   [u]         [v-
 1]   |=2,m[u][v-1] &   48?W][v-1   ] &   15]]):0:0;u?m[u   -1][v]|=1   ,m[
 u-               1][   v]&         48?               W-1   ][v         ]&
15]   ]):0:0;v<   255   ?m[   u][v+1]|=8,m[u][v+1]&   48?   W][   v+1]&15]]
):0         :0;         u <               255   ?m[   u+1         ][v   ]|=
4,m[u+1][   v]&48?W+1][v]&15]]):0:0;W][   v]&   15]   ]);}cu(char*q){   return
 *q               ?cu   (q+         1)&         1?q   [0]               ++:
q[0   ]--   :1;   }d(   int   u ,   int/**/v,   int/**/x,   int   y){   int
Y=y   -v,   X=x         -u;   int         S,s   ;Y<         0?Y   =-Y   ,s,
s=-   1:(   s=1);X<0?X=-X,S   =-1  :(S=   1);   Y<<=   1;X<<=1;   if(X>Y){
int   f=Y               -(X   >>1   );;               while(u!=         x){
f>=   0?v+=s,f-=X:0;u   +=S   ;f+=   Y;m[u][v]|=32;mvwaddch(w,v   ,u,   m[u
 ][               v]&   64?   60:         46)         ;if         (m[   u][
v]&16){c(u,v);;   ;;;   ;;;   return;}}   }else{int   f=X   -(Y>>1);;   while
 (v   !=y         ){f   >=0         ?u   +=S,               f-=         Y:0
 ;v   +=s   ;f+=X;m[u][v]|=   32;mvwaddch(w,v   ,u,m[u][v]&64?60:46);if(m[u
 ][                     v]&         16)   {c(   u,v                     );
  ;   return;;;}}}}Z(   int/**/a,   int   b){   }e(   int/**/y,int/**/  x){
int               i ;         for         (i=         a;i               <=a
+S;i++)d(y,x,i,b),d(y,x,i,b+L);for(i=b;i<=b+L;i++)d(y,x,a,i),d(y,x,a+   S,i
 );                     ;;;         ;;;         ;;;               ;;;   ;
  mvwaddch(w,x,y,64);   ;;;   ;;;   ;;;   prefresh(   w,b,a,0,0   ,L-   1,S-1
);}             main(         int               V ,   char              *C[
  ]   ){FILE*f=   fopen(V==1?"arachnid.c"/**/   :C[   1],"r");int/**/x,y,c,
                (source code cut here)
```

Screenshoot



Recreational Obfuscation: Phillips entry of IOCCC'88

Program code

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Output

```
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

On the fourth day of Christmas my true love gave to me
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the fifth day of Christmas my true love gave to me
```

Output (cont)

```
On the eighth day of Christmas my true love gave to me
eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the ninth day of Christmas my true love gave to me
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the tenth day of Christmas my true love gave to me
ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
```

Bad Style Coding as an Art

Another example

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l              ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

Bad Style Coding as an Art

Another example: Computing Integer Square Roots

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

It actually works

```
$ ./cheong 1234
35
```

($35 \times 35 = 1225$; $35 \times 36 = 1296$)

```
$ ./cheong 112233445566
335012
```

$335012 \times 335012 = 112233040144$

$335013 \times 335013 = 112233710169$

Bad Style Coding as an Art

Another example: Computing Interger Square Roots

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

It actually works

```
$ ./cheong 1234
35
```

$(35 \times 35 = 1225;\ 35 \times 36 = 1296)$

```
$ ./cheong 112233445566
335012
```

$335012 \times 335012 = 112233040144$

$335013 \times 335013 = 112233710169$

Author claim: code self-documented...

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

Bad Style Coding as an Art

Another example: Computing Interger Square Roots

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

It actually works

```
$ ./cheong 1234
35
```

$(35 \times 35 = 1225;\ 35 \times 36 = 1296)$

```
$ ./cheong 112233445566
335012
```

$335012 \times 335012 = 112233040144$

$335013 \times 335013 = 112233710169$

Author claim: code self-documented...

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l                ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

> *It is an old observation that the best writers sometimes disregard the rules of rhetoric. When they do so, however, the reader will usually find in the sentence some compensating merit, attained at the cost of the violation.* **Unless he is certain of doing as well, he will probably do best to follow the rules**.
>
> – William Strunk, Jr. (1918)

Last one, just for fun: dhyang IOCCC'00

Saitou Hajime image

Source code

```
                #define/**/X
               char*d="X0[!4cM,!"
             "4cK`*!4cJc(!4cHg&!4c$j"
           "8f'!&~]9e)!'|:d+!)rAc-!*m*"
         ":d/!4c(b4e0!lr2e2!/t0e4!-y-c6!"
       "+|,c6!)f$b(h*c6!(d'b(i)d5!(b*a``&c"
       ")c5!'b+`&b'c)c4!&b-_$c'd*c3!&a.h'd+"
     "dl!%a/g'e+eO!%b-g(d.d/!&c*h'dld-!(d%g)"
   "d4d+!*l,d7d)!,h-d;c'!.bOc>d%!A`Dc$![7)35E"
  "!'1cA,,!2kE`*!-s@d(!(k(f//g&!)f.e5'f(!+a+)"
  "f%2g*!?f5f,!=f-*e/!<d6el!9eO'f3!6f)-g5!4d*b"
  "+e6!Of%k)d7!+~^'c7!)z/d-+!'n%aO(d5!%cla+/d4"
  "!2)c9e2!9b;e1!8b>e/!     7cAd-!5fAe+!7fBe(!"
 "8hBd&!:iAd$![7S,QO!l     bF 7!lb?'_6!lc,8b4"
 "!2b*a,*d3!2n4f2!${4    f.      '!%y4e5!&f%"
 "d-^-d7!4c+b)d9!4c-a    'd        :!/i(``&d"
 ";!+l'a+d<!)l*b(d=!'   m-        a  &d>!&d'"
 "`O_&c?!$dAc@!$cBc@!$   b         <    ^&d$`"
 ":!$d9_&l++^$!%f3a'    nl              $ !&"
 "f/c(o/_%!(f+c)q*c     %!         *       f &d+"
 "f$s&!-n,d)n(!Oi-     c-         k)       !  3d"
 "/bOh*!H`7a,![7*     i]          5        4   71"
 "[=ohr&o*t*q*`*d      *v         *r         ;  O2"
 "7*~=h./}tcrsth      &t          :          r   9b"
"]..b-725-.t--//      #r         [           <   t8-"
"752793?  <.~;b      ].t--+r     /           #    53"
"7-r[/9~X  .v9O      <6/<.v;-52/={            k   goh"
"./}q;   u  vto     hr  `.i*$engt$            $    ,b"
";$/     =t ;v;     6     =`it.`;7=`          :    ,b-"
"725    = / o`.    .d       ;b]`--[/+       55/    }o"
"`.d   :   - ?5   /           }o`.'     v/i]q      - "
"-[;   5  2  =`  it            .        o;53-      . "
"v96   <7 /      =o            :           d        =o"
"--/i  ]q--      [;           h.           /        = "
"i]q--[  ;v      9h           ./           <        - "
"52={cj   u      c&`          i   t       . o       ; "
"?4=o:d=         o--          /  i        ]q        - "
"-[;54={  cj     uc&          i]q          -         -"
"[;76=i]q[;6     =vsr        u.i           /         ={"
"=),BihY_gha     ,)\0        "             ,         o [
 3217];int i,   r,w,f        ,             b        ,x ,
 p;n(){return   r  <X        X              X       X  X
 768?d[X(143+   X  r++       +              *d      )  %
  768]:r>2659   ?  59:       (               x      =  d
  [(r++-768)%   X  947      +             768]      ) ?
  x^(p?6:0):(p  =   34      X           X           X )
  ;}s(){for(x=  n   ();     (           x^          ( p
 ?6:0))==32;x=  n    ()     )   ;return x           ; }
 void/**/main X      ()     {           r          =  p
 =0;w=sprintf  (X     X     X         X X          X o
 ,"char*d=");  for          (    f=1;f <           * d
 +143;)if(33-(  b=d         [      f++ X          ]  )
 ){if(b<93){if   X(!        p          )            o
  [w++]=34;for    X(i       =         35            +
   (p?0:1);i<b;    i++      )         o
   [w++]=s();o[     w++               ]
    =p?s():34;}     else              X
      {for(i=92;     i<b;            i
       ++)o[w++]=     32;}           }
            else o     [w++         ]
                       =10;o        [
                         w]=0     ;
                          puts(o);}
```

Last one, just for fun: dhyang IOCCC'00

Saitou Hajime image that prints a prog

Source code

```
                #define/**/X
              char*d="X0[!4cM,!"
            "4cK`*!4cJc(!4cHg&!4c$j"
          "8f'!&~]9e)!']:d+!)rAc-!*m*"
        ":d/!4c(b4e0!1r2e2!/t0e4!-y-c6!"
      "+|,c6!)f$b(h*c6!(d'b(i)d5!(b*a``&c"
      ")c5!'b+`&b'c)c4!&b-_$c'd*c3!&a.h'd+"
    "dl!%a/g'e+e0!%b-g(d.d/!&c*h'd1d-!(d%g)"
  "d4d+!*l,d7d)!,h-d;c'!.b0c>d%!A`Dc$![7)35E"
 "!'1cA,,!2kE`*!-s@d(!(k(f//g&!)f.e5'f(!+a+)"
 "f%2g*!?f5f,!=f-*e/!<d6el!9e0'f3!6f)-g5!4d*b"
  "+e6!0f%k)d7!+~^'c7!)z/d-+!'n%aO(d5!%c1a+/d4"
   "!2)c9e2!9b;el!8b>e/!     7cAd-!5fAe+!7fBe(!"
  "8hBd&!:iAd$![7S,QO!1     bF 7!1b?'_6!1c,8b4"
   "!2b*a,*d3!2n4f2!${4    f.      '!%y4e5!&f%"
   "d-^-d7!4c+b)d9!4c-a     'd        :!/i(``&d"
   ";!+l'a+d<!)l*b(d=!'    m-        a  &d>!&d'"
   "`O_&c?!$dAc@!$cBc@!$   b         <    ^&d$`"
   ":!$d9_&l++^$!%f3a'     n1               $ !&"
   "f/c(o/_%!(f+c)q*c      %!               f &d+"
   "f$s&!-n,d)n(!Oi-       c-          k)   !  3d"
   "/bOh*!H`7a,![7*      i]            5     4  71"
  "[=ohr&o*t*q*``*d       *v           *r      ;  O2"
  "7*~=h./}tcrsth       &t             :       r  9b"
 "],.,b-725-.t--//      #r             [       <  t8-"
 "752793?  <.~;b       ].t--+r        /       #   53"
 "7-r[/9~X  .v90       <6/<.v;-52/={            k   goh"
 "./}q;    u  vto     hr  `.i*$engt$            $    ,b"
 ";$/      =t ;v;     6     =`it.`;7=`          :    ,b-"
 "725     = / o`.     .d       ;b]`--[/+       55/    }o"
 "`.d   :    - ?5    /           }o`.'     v/i]q      - "
 "-[;   5  2  =`  it              .       o;53-       . "
 "v96   <7 /       =o             :           d        =o"
 "--/i  ]q--       [;            h.           /        = "
 "i]q--[  ;v       9h            ./           <        - "
 "52={cj   u       c&`           i   t        . o       ; "
 "?4=o:d=          o--           /  i        ]q        - "
 "-[;54={  cj      uc&           i]q           -         -"
 "[;76=i]q[;6      =vsr          u.i           /         ={"
 "=),BihY_gha      ,)\0          "             ,          o [
 3217];int i,    r,w,f          ,              b         ,x ,
 p;n(){return    r  <X          X              X        X  X
 768?d[X(143+    X  r++         +              *d       )  %
  768]:r>2659    ?  59:         (              x        =  d
  [(r++-768)%    X  947       +              768]       ) ?
  x^(p?6:0):(p   =   34       X             X           X )
  ;}s(){for(x=   n   ();      (            x^           ( p
 ?6:0))==32;x=   n    ()     )   ;return x              ; }
 void/**/main X       ()     {             r           =  p
 =0;w=sprintf  (X     X      X         X X             X o
 ,"char*d=");  for           (    f=1;f <              * d
 +143;)if(33-(  b=d          [      f++ X             ]  )
 ){if(b<93){if    X(!        p           )              o
  [w++]=34;for     X(i       =          35              +
   (p?0:1);i<b;    i++       )          o
   [w++]=S();o[    w++                  ]
    =p?s():34;}    else                 X
      {for(i=92;    i<b;                i
       ++)o[w++]=     32;}              }
            else o     [w++            ]
                       =10;o           [
                         w]=0      ;
                          puts(o);}
```

Output 1

```
char*d=")35E!'lcA,,!""2kE`*!-s@d(!(k(f//g&!)f.e5'f(!+a+)f%2g*!?f5f,!=f-*e/!<d"
"6el!9e0'f3!6f)-g"      "5!4d*b+e6!0f%k)d7!+~^'c7!)z/d-+!'n%aO(d5!""%c1a+/d4"
"!2)c9e2!9b;el!8b"      ">e/!7cAd-!5fAe+!7fBe(!8hBd&!:iAd$![7S,Q"    "O!1bF7"
"!1b?'_6!1c,8b4!2"      "b*a,*d3!2n4f2!${4f.'!%y4e5!&f%d-^-d7!4"       "c+b)"
"d9!4c-a'd:!/i(``"      "&d;!+l'a+d<!)l*b(d=!'m-a&d>!&d'`0_&c"          "?!"
"$dAc@!$cBc@!$b<^"      ""              "&d$`:!$d9_&l++^$!%f3"          "a'n"
"1_"                                  "$!&f/c(o/_%!(f+c)q*c%"         "!*f&d"
"+f$"                                "s&!-n,d)n(!Oi-c-k)!3d/"       "bOh*!H`"
"7a,![                          "7XO[!4cM,!4cK`*!4cJc(!4cHg&!4"       "c$j8f'!&~"
"]9e)!'"                "|;d+!)""rAc-!*m*:d/!4c(b4e0!1r"       "2e2!/t0e4!-"
"y-c6!+|,c6!)f$b("      "h*c6"      "!(d'b(i)d5!(b*a'"          "`&c)c5!'b+`&"
"b'c)c4!&b-_$c"                     "'d*c3!&a.h'd+d"         "1!%a/g'e+e0!%b"
"-g(d.d/!&c*"                       "h'd1d-!(d%g)"        "d4d+!*l,d7d)!,h-"
"d;c'!.b0c"                            ">d%!A`Dc$"       "![7*i]5471[=ohr&o*"
"t*q*`*d"                               "*v*r;027"       "*~=h./}tcrsth&t:r9"
"b]..,b"          ""      "-725"          "-.t--/"       "/#r[<t8-752793?<.~"
";b].`"        "t--"      "+r/#"          "537-r"         "[/9~X.v90<6/<.v;-"
"52/="      "{kgoh."     "/}q"    ";u"      "vtohr"       "`.i*$engt$$,b;$/="
"t;v"       ";6=`it."   "`;"     "7=`"       ":,b-7"       "25=/o`..d;b]`--["
"/+"      "55/}o`.d:"   ""       "-?5"        "/}o`.'"       "v/i]q--[;52=`it"
".o"     ";53-.v96<7"            "/=o"        ":d=o--/"        "i]q--[;h./=i]"
"q"     "--[;v9h./<-"          "52="         "{cjuc&`"         "it.o;?4=o:d="
"o"      "--/i]q--["          ";54="        "{cjuc&i]q-"        "-[;76=i]q[;"
"6="       "vsru.i"           "/={"        "=),BihY_gha,)"        "",o[3217];
int i,         r,w          ,f,b,p,        t=641,x;n(){return         r<t?d[(*
d+143+(r                ++))%t]:r>       +1341?59:(x=d[(r++-t)       %351+t]
)?x^(p?6:0          ):(p=+34);}        main(){w=sprintf(o,"char"       "*d="
);r=p=0;for(      f=1;f<*d+143;)       if((b=d[f++])-33){if(b<+93){        if(
!p)o[w++]=34;for(i=35+(p?0:1);i    <b;i++)o[w++]=n();o[w++]=p?n():+34      ;}
else for(i=92;i<b;i++)o[w++]=32;}else o[w++]=10;o[w]=0;puts(o);};/*Don_Yang*/;
```

Last one, just for fun: dhyang IOCCC'00

Saitou Hajime image that prints a prog that prints a prog

Source code

Output 1

Output 2

Last one, just for fun: dhyang IOCCC'00

Saitou Hajime image that prints a prog that prints a prog that prints a prog

Source code

```
                #define/**/X
              char*d="X0[!4cM,!"
            "4cK`*!4cJc(!4cHg&!4c$j"
          "8f'!&~]9e)!']:d+!)rAc-!*m*"
        ":d/!4c(b4e0!1r2e2!/t0e4!-y-c6!"
      "+|,c6!)f$b(h*c6!(d'b(i)d5!(b*a`&c"
      ")c5!'b+`&b'c)c4!&b-_$c'd*c3!&a.h'd+"
    "dl!%a/g'e+e0!%b-g(d.d/!&c*h'd1d-!(d%g)"
   "d4d+!*l,d7d)!,h-d;c'!.b0c>d%!A`Dc$![7)35E"
  "!'1cA,,!2kE`*!-s@d(!(k(f//g&!)f.e5'f(!+a+)"
  "f%2g*!?f5f,!=f-*e/!<d6e1!9e0'f3!6f)-g5!4d*b"
  "+e6!Of%k)d7!+~^'c7!)z/d-+!'n%a0(d5!%c1a+/d4"
  "!2)c9e2!9b;el!8b>e/!     7cAd-!5fAe+!7fBe(!"
 "8hBd&!:iAd$![7S,QO!l     bF 7!lb?'_6!1c,8b4"
 "!2b*a,*d3!2n4f2!${4    f.      '!%y4e5!&f%"
 "d-^-d7!4c+b)d9!4c-a    'd        :!/i(``&d"
 ";!+l'a+d<!)l*b(d=!'   m-         a  &d>!&d'"
 "`O_&c?!$dAc@!$cBc@!$   b          <    ^&d$`"
 ":!$d9_&l++^$!%f3a'     nl               $ !&"
 "f/c(o/_%!(f+c)q*c     %!               f &d+"
 "f$s&!-n,d)n(!0i-      c-          k)      !  3d"
 "/bOh*!H`7a,![7*     i]            5       4   71"
"[=ohr&o*t*q*`*d      *v           *r         ;  02"
"7*~=h./}tcrsth       &t           :          r   9b"
"]..b-725-.t--//      #r           [          <   t8-"
"752793?  <.~;b       ].t--+r      /          #    53"
"7-r[/9~X  .v90      <6/<.v;-52/={               k   goh"
"./}q;    u  vto     hr   `.i*$engt$             $    ,b"
";$/      =t ;v;     6     =`it.`;7=`            :    ,b-"
"725     = / o`.     .d       ;b]`--[/+       55/     }o"
"`.d   :    - ?5    /           }o`.'     v/i]q       - "
"-[;   5  2  =`  it              .        o;53-        . "
"v96   <7 /       =o             :           d        =o"
"--/i  ]q--       [;             h.          /        = "
"i]q--[  ;v       9h             ./          <        - "
"52={cj   u       c&`            i   t       . o      ; "
"?4=o:d=          o--            /  i        ]q       - "
"-[;54={  cj      uc&            i]q         -        -"
"[;76=i]q[;6      =vsr           u.i         /        ={"
"=),BihY_gha      ,)\0           "           ,        o [
 3217];int i,     r,w,f          ,           b        ,x ,
 p;n(){return     r  <X          X           X        X  X
 768?d[X(143+     X  r++         +           *d       )  %
  768];r>2659     ?  59:         (           x        =  d
  [(r++-768)%     X  947       +               768]     ) ?
  x^(p?6:0):(p    =   34       X             X          X )
  ;}s(){for(x=    n   ();      (           x^           ( p
 ?6:0))==32;x=    n     ()     )   ;return x            ; }
 void/**/main X         ()     {           r          =  p
 =0;w=sprintf   (X      X      X         X X          X o
 ,"char*d=");   for             (     f=1;f <          * d
 +143;)if(33-(  b=d             [       f++ X        ]  )
 ){if(b<93){if    X(!           p           )          o
  [w++]=34;for     X(i          =           35         +
   (p?0:1);i<b;     i++         )           o
   [w++]=s();o[     w++                     ]
    =p?s():34;}     else                    X
      {for(i=92;     i<b;                   i
       ++)o[w++]=      32;}                  }
            else o     [w++                  ]
                       =10;o                 [
                         w]=0      ;
                          puts(o);}
```

Output 1

```
char*d=")35E!'lcA,,!""2kE`*!-s@d(!(k(f//g&!)f.e5'f(!+a+)f%2g*!?f5f,!=f-*e/!<d"
"6e1!9e0'f3!6f)-g"      "5!4d*b+e6!0f%k)d7!+~^'c7!)z/d-+!'n%a0(d5!""%c1a+/d4"
"!2)c9e2!9b;el!8b"      ">e/!7cAd-!5fAe+!7fBe(!8hBd&!:iAd$![7S,Q"    "O!lbF7"
"!lb?'_6!1c,8b4!2"      "b*a,*d3!2n4f2!${4f.'!%y4e5!&f%d-^-d7!4"       "c+b)"
"d9!4c-a'd:!/i(``"      "&d;!+l'a+d<!)l*b(d=!'m-a&d>!&d'`O_&c"          "?!"
"$dAc@!$cBc@!$b<^"      ""          "&d$`:!$d9_&l++^$!%f3"          "a'n"
"l_"                                "$!&f/c(o/_%!(f+c)q*c%"          "!*f&d"
"+f$"                               "s&!-n,d)n(!0i-c-k)!3d/"       "b0h*!H`"
"7a,![                    "7X0[!4cM,!4cK`*!4cJc(!4cHg&!4"       "c$j8f'!&~"
"]9e)!'"                "|:d+!)""rAc-!*m*:d/!4c(b4e0!1r"       "2e2!/t0e4!-"
"y-c6!+|,c6!)f$b("      "h*c6"       "!(d'b(i)d5!(b*a'"          "&c)c5!'b+`&"
"b'c)c4!&b-_$c"                      "'d*c3!&a.h'd+d"          "l!%a/g'e+e0!%b"
"-g(d./!&c*"                         "h'd1d-!(d%g)"          "d4d+!*l,d7d)!,h-"
"d;c'!.b0c                              ">d%!A`Dc$"       "![7*i]5471[=ohr&o*"
"t*q*`*d                                "*v*r;027"       "*~=h./}tcrsth&t:r9"
"b]..b"          ""      "-725"           "-.t--/"       "/#r[<t8-752793?<.~"
";b]."          "t--"    "+r/#"           "537-r"        "[/9~X.v90<6/<.v;-"
"52/="      "{kgoh."     "/}q"     ";u"      "vtohr"       "`.i*$engt$$,b;$/="
"t;v"         ";6=`it."  ";"       "7=~"      ":,b-7"      "25=/o`..d;b]`--["
"/+"       "55/}o`.d:"   ""        "-?5"       "/}o`.'"      "v/i]q--[;52=`it"
".o"        ";53-.v96<7"           "/=0"        ":d=o--/"      "i]q--[;h./=i]"
"q"      "--[;v9h./<-"           "52="          "{cjuc&`"       "it.o;?4=o:d="
"o"        "--/i]q--["           ";54="         "{cjuc&i]q-"      "-[;76=i]q[;"
"6="       "vsru.i"              "/={"          "=),BihY_gha,)"       "",o[3217];
int i,          r,w          ,f,b,p,        t=641,x;n(){return          r<t?d[(*
d+143+(r              ++))%t]:r>       +1341?59:(x=d[(r++-t)        %351+t]
)?x^(p?6:0          ):(p=+34);}       main(){w=sprintf(o,"char"       "*d="
);r=p=0;for(      f=1;f<*d+143;)       if((b=d[f++])-33){if(b<+93){       if(
!p)o[w++]=34;for(i=35+(p?0:1);i    <b;i++)o[w++]=n();o[w++]=p?n():+34      ;}
else for(i=92;i<b;i++)o[w++]=32;}else o[w++]=10;o[w]=0;puts(o);};/*Don_Yang*/;
```

Output 2

```
char*d="S,QO!lbF7!lb?'_""6!1c,8b4!2b*a,*d3""!2n4f2!${4f.'!%y4e5!&f%d-^-d7!4c+"
"b)d""9!4c-a'd:!/i("      "'`&d;!+l'a+d<!)l*b(d=!'m-a&d>""!&d``O_&""c?!$dAc@"
"!$cBc@!$b<^&d$"            "`:!$d9_&l++^$!%f3a'nl_$!&f/c(o/_%"    "!(f+c)"
"q*c%!*f&d"                          "+f$s&!-n,d)n(!0i-c-k)!3d/b0h"          "*!H`"
"7a,!"                    "[!4cM,!4cK`""*!4cJc(!4cH"          "g&"
"!4c$j"          "8f'!&~]9e)"        "!']:d+!)rAc-!*m*:""d/!"          "4c(b"
"4e0!1r2"     "e2!/t0e""4!-y-"          "c""6!+|,c6!)f$b(h"          "*c6!(d"
"'b(i)d5!(b*a'`&c)c5!'b+`&b'"          "c)c4!&b-_$c'd*c3!"          "&a.h'd+d"
"l!%a/g'e+e0!%b-g(d.d/!&c*"          "h'd1d-!(d""%g)d4d"          "+!*l,d7d)!,"
"h-d;c'!.b0c>d%!A`Dc$![7)"          "35E!'lcA,,!2kE`*!-"          "s@d(!(k(f//g&"
"!)f.e5'f(!+a+)f%2g*!?"          "f5f,!=f-*e/!""<d6"          "el!9e0'f3!6f)-g"
"5!4d*b+e6!0f%k)d7!"          "+~^'c""7!)z/d-+!"          "'n%a0(d5!%c1a+/d4"
"!2)c9e2!9b;el!8b"          ">e/!7c"       "Ad-!5fA"          "e+!7fBe(!8hBd&!:iA"
"d$![7*i]5471"          "["          "=ohr&"          "o*t*q*`*d*v*r;027*~"
"=h./}tc"                          "rst"          "h&t:r9b]..b-725-.t-"
"-//#r"                          "[<t8-752793"          "?<.~;b].t""--+r/#5"
"37-"                  "r"     "[/9~X.v90<6/""<.v;"          "-52/={kgoh./}q;uv"
"t"          "ohr`.i*$engt$"     "$,b;$/=""t;v;6=`it.`"          ";7=`:,b-725=/o`."
".d;b]`--[/+55/""}o`.d"          ":-?5/}o`.'v/i]q--[;52"          "=`it.o;53-.v96"
"<7/=o:d=o--/i]q--[;h."          "/=i]q--[;v9h./<-52={cju"          "c&`it.o;?4=o:"
"d=o--/i]q--[;54={cju"          "c&i]q--[;76=i]q[;6=vsru.i/"          "={=),BihY_g"
"ha,)",o[3217];int i,          r,w,f,b,p,t=641,x;n(){return r<          t?d[(*d+143"
+(r++))%t]:r>+1341?          59:(x=d[(r++-t)%351+t])?x^(p?6:          0):(p=+34
);}main(){w=sprintf(o          ,"char""*d=");r=p=0;for(f=1;f<*d          +143;)
if((b=d[f++])-33){if(b          <+93){if(!p)o[w++]=34;for(i=35+(          p?0:
1);i<b;i++)o[w++]=n();o[          w++]=p?n():+34;}else for(i=92;i          <b
;i++)o[w++]=32;}else o[w++]=10;o[w]=0;puts(o);};/*Don_Yang*/;;;;;;;;;;;;;;;;;
```

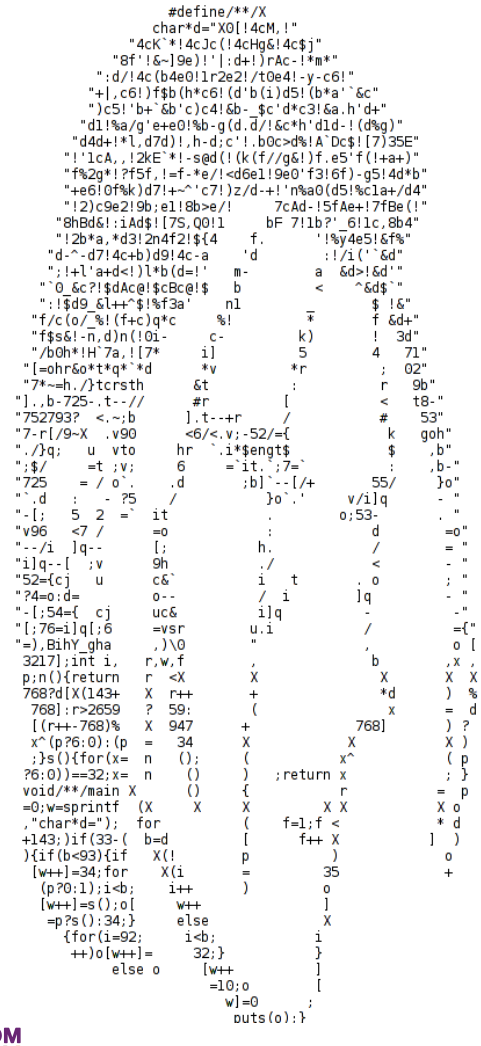
Output 3

```
char*d="X0[!4cM,""!4cK`*!4cJc(!4cHg&!4c$j8f'!&~]9e)!']:d+!)rAc-""!*m*:d/!4c(b"
"4e0!1r2e2!/t0"      "e4!-y-c6!+|,c6!)f$b(h*c6!(d'b(i)d5""!(b*a'`&c)c5!'b+`&b"
"'c)c4!&b-_$c'"      "d*c3!&a.h'd+d1!%a/g'e+e0!%b""-g("   "d./!&c*h'd1d-!(d%"
"g)d4d+!*l,d7d"      ")!,h-d;c""'!.b0c>d%!A`Dc$![7)3"      "5E!'lcA,,!2kE`*!"
"-s@d(!(k(f//g&"      "!)f.e5"      "'f(!+a+)""f%2g*!"          "?f5f,!=f-*e/!<d"
"6el!9e0'f3!6f)"                  "-g5!4d*b+e6!0f%k"          ")d7!+~^'c7!)z/"
""                              "d-+!'n%a0(d5!%c1"          "a+/d4!2)c9""e2!"
"9"                           "b;el!8b>e/!7cAd-"          "!5fAe+!7fBe(!8hBd"
"&!"          ":"       "iAd$![7S,"   "QO!lbF7!1"          "b?'_6!1c,8b4!2b*a,*"
"d3!2n4f2!${4f.'!"       "%y4e5!&"      "f%d-^"          "-d7!4c+b)d9!4c-a'd:!/"
"i(``&d;!+l'a+d<!"       ")l*b(d=!'"     "m-a"          "&d>!&d'`O_&c?!$dAc@!$c"
"Bc@!$b<^&d$"             "`:!$""d9_"    "&l"          "++^$!%f3a'nl_$!&f/c(o/_"
"%!(f+c)"                 "q*c"      "%!*f&d+"          "f$s&!-n,d)n(!0i-c-k)!3d/"
"b0h*!"                 "H`7a,!"       "[7*i"          "]5471[=ohr&o*t*q*`*d*v*r;"
"027"                  "*~=h./}tc"     "rs"          "th&t:r9b]..b-725-.t--//#r["
"<t"          "8-7"     "52793?<.~;b]"    ".t"          "--+r/#537-r[/9~X.v90<6/<.v;"
""          "-52/={kgoh./}q;uvtohr`.i*$eng"          "t$$,b;$/=t;v;6=`it.`;7=`:,b-"
""        "725=/o`..d;b]`--[/+55/}o`.d:-?"          "5/}o`.'v/i]q--[;52=`it.o;53-"
""      ".v96<7/=o:d=o--/i]q--[;h"   "./"          ""      "=i]q--[;v9h./<-52={cju"
""        "c&`it.o;?4=o:d=o--/i]"     "q-"          "-[;54={""cjuc&i]"   ""
"q"           "--[;76=i]q[;6=v"       "sru"          ".i/={=),BihY_"    ""
"gh"          "a,)",o[3217]       ;int i          ,r,w,f,b,p,t=   641
,x;n()          {return r       <t?d[(*          d+143+(r       ++)
)%t]:r>+          1341       ?59:(x=d[          (r          ++-t
)%351+t])?x          ^(p?6:0):(       p=+34);          }main(
){w=sprintf(o,          "char""*d="      );r=p=0;for          (f=1;f<
*d+143;)if((b=d[f       ++])-33){if(b      <+93){if(!p)o[          w++]=34;
for(i=35+(p?0:1);i<b;i++)o[w++]=n();o[    w++]=p?n():+34;}else      for(i=92;i
<b;i++)o[w++]=32;}else o[w++]=10;o[w]=0;puts(o);};/*Don_Yang*/;;;;;;;;;;;;;;;;;
```

Last one, just for fun: dhyang IOCCC'00

Saitou Hajime image that prints a prog that prints a prog that prints a prog . . .
Repeating endlessly "aku soku zan", Hajime's motto meaning *slay evil imediatly*.

Source code



Output 1

Output 2

Output 3

Output 4 (=1)

Your first C program

The classical Hello World

hello.c

```
#include <stdio.h>
int main(int argc, char *argv[]){
  printf("Hello, world\n");
}
```

For the record: same in Java

hello.c

```
class HelloWorld {
 public static void main(String[]arg){
  System.out.println("Hello, world");
}}
```

Compile and run it

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

Compiling and running Java code

```
$ javac HelloWorld.java
$ java -cp . HelloWorld
```

Explanations

- `#include` can be seen as the equivalent of `import` directives
- `main` is the *entry point* of every program (same in C and Java)

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

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

Conclusion on C (for now)

C is the modern assembly language

- It's quite prehistorical
  - Compilation process not trivial (even with only one file)
  - Cryptic error messages
  - No fancy stuff in standard library
- Programs can be really fast
  - If you do them right; easy to code slow C programs too
- You have the full power of doing everything
  - No matter what you want to code, it's possible in C
  - A lot of code were already developed in C (check `koders.com`)
  - C poses no rule to limit your imagination...
  - ...but there is no barrier to prevent you doing stupid things

You need to master C to understand your machine

- The operating system is in C, just like the virtual machines
- And then, you're free to use it or not
  Depending on whether you're seeking for fast programs or fast coding

Chapter 1

C and Unix

- Introduction
  - C? UNIX? What is all this about?
  - Why do we need to study C?
  - Why do we need to study C and UNIX together?
- C as Second Language
  - C vs. Java
  - How to survive in C?
  - Your first C program
- First steps in Unix
  - Désignation des fichiers
  - Protection des fichiers
  - Using the terminal

First steps in Unix

This OS gives a central role to **files**

- Contains data and executable programs (quite usual)
- Communication with user : config files, `stdin`, `stdout`
- Communication between processes: sockets, pipes, etc.
- Interface to the kernel: `/proc`
- Interface to the hardware: peripheral in `/dev`

First steps in Unix

This OS gives a central role to **files**

- Contains data and executable programs (quite usual)
- Communication with user : config files, `stdin`, `stdout`
- Communication between processes: sockets, pipes, etc.
- Interface to the kernel: `/proc`
- Interface to the hardware: peripheral in `/dev`

The **Terminal** is an interface of choice

- Graphical interfaces exist too, but I still prefer the terminal
- Lots of tricks make you more efficient with the terminal (more button on my keyboard than on my mouse)
- If you can't do it in one step, type a one-line script directly

First steps in Unix

This OS gives a central role to **files**

- Contains data and executable programs (quite usual)
- Communication with user : config files, `stdin`, `stdout`
- Communication between processes: sockets, pipes, etc.
- Interface to the kernel: `/proc`
- Interface to the hardware: peripheral in `/dev`

The **Terminal** is an interface of choice

- Graphical interfaces exist too, but I still prefer the terminal
- Lots of tricks make you more efficient with the terminal (more button on my keyboard than on my mouse)
- If you can't do it in one step, type a one-line script directly

Read The Fine Manual (RTFM)

- The command `man` gives you access to a large corpus of knowledge
- `man prog` or `man function` $\rightsquigarrow$ documentation of that program or function

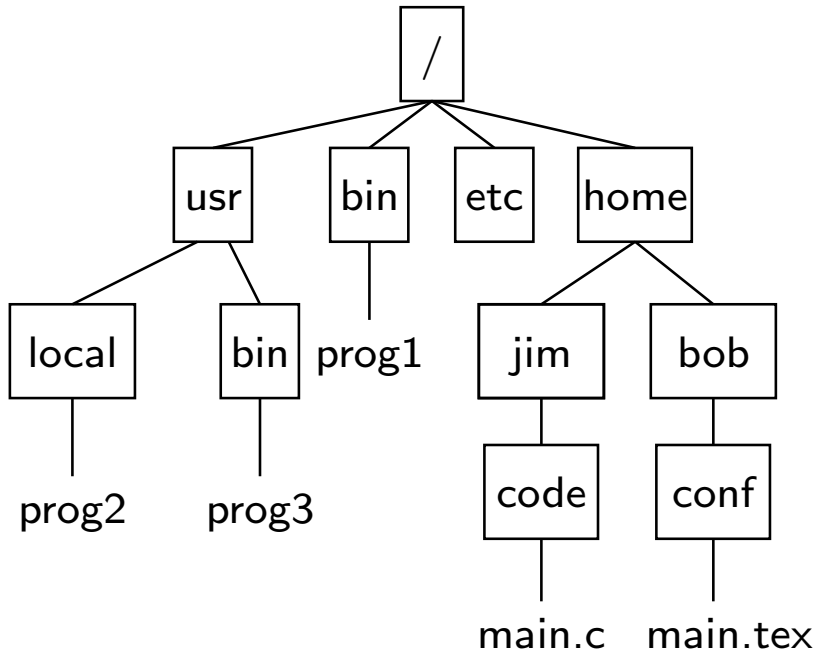
Désignation des fichiers

Désignation symbolique (nommage): Organisation hiérarchique

- Noeuds intermédiaires: répertoires (*directory* – ce sont aussi des fichiers)
- Noeuds terminaux: fichiers simples
- Nom absolu d'un fichier: le chemin d'accès depuis la racine

Exemples de chemins absolus :

```
/
/bin
/usr/local/bin/prog
/home/bob/conf/main.tex
/home/jim/code/main.c
```



Raccourcis pour simplifier la désignation

Noms relatifs au répertoire courant

- Depuis `/home/bob`, `conf/main.tex` = `/home/bob/conf/main.tex`

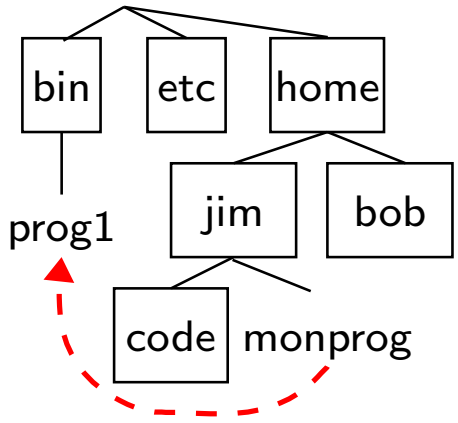
Abréviations

- Répertoire père: depuis `/home/bob`, `../jim/code` = `/home/jim/code`
- Répertoire courant: depuis `/bin`, `./prog1` = `/bin/prog1`
- Depuis n'importe où, ~bob/ = /home/bob/ et ~/ = /home/<moi>/

Liens symboliques

Depuis `/home/jim`

- Création du lien: `ln -s` *cible nom_du_lien*
  Exemple: `ln -s /bin/prog1 monprog`
- `/home/jim/prog1` désigne `/bin/prog1`
- Si la cible est supprimée, le lien devient invalide



Règles de recherche des exécutables

- Taper le chemin complet des exécutable (/usr/bin/ls) est lourd
- $\Rightarrow$ on tape le nom sans le chemin et le shell cherche
- Variable environnement PATH: liste de répertoires à examiner successivement
  `/usr/local/bin:/usr/local/sbin:/sbin:/usr/sbin:/bin:/usr/bin:/usr/bin/X11`
- Commande `which` indique quelle version est utilisée

Exercice : Comment exécuter un script nommé `gcc` dans le répertoire courant?

- Solution 1:
- Solution 2:

Utilisations courantes des fichiers

- Unix: fichiers = suite d'octets sans structure interprétée par utilisateur
- Windows: différencie fichiers textes (où \n est modifié) des fichiers binaires

Programmes exécutables

- Commandes du système ou programmes créés par un utilisateur
- Exemple: `gcc -o test test.c ; ./test`
- Question: pourquoi `./test` ?

Fichiers de données

- Documents, images, programmes sources, etc.
- Convention:

  Exemples : .c (programme C), .o (binaire translatable, cf. plus loin), .h (entête C), .gif (un format d'images), .pdf (Portable Document Format), *etc.*
  Remarque: ne pas faire une confiance aveugle à l'extension (*cf.* `man file`)

Fichiers temporaires servant pour la communication

- Ne pas oublier de les supprimer après usage
- On peut aussi utiliser des tubes (*cf.* RS l'an prochain)

Utilisations courantes des fichiers

- Unix: fichiers = suite d'octets sans structure interprétée par utilisateur
- Windows: différencie fichiers textes (où \n est modifié) des fichiers binaires

Programmes exécutables

- Commandes du système ou programmes créés par un utilisateur
- Exemple: `gcc -o test test.c ; ./test`

  Produit programme exécutable dans fichier `test`; exécute le programme test
- Question: pourquoi `./test` ?

Fichiers de données

- Documents, images, programmes sources, etc.
- Convention:

  Exemples : .c (programme C), .o (binaire translatable, cf. plus loin), .h (entête C), .gif (un format d'images), .pdf (Portable Document Format), *etc.*
  Remarque: ne pas faire une confiance aveugle à l'extension (*cf.* `man file`)

Fichiers temporaires servant pour la communication

- Ne pas oublier de les supprimer après usage
- On peut aussi utiliser des tubes (*cf.* RS l'an prochain)

Utilisations courantes des fichiers

- Unix: fichiers = suite d'octets sans structure interprétée par utilisateur
- Windows: différencie fichiers textes (où \n est modifié) des fichiers binaires

Programmes exécutables

- Commandes du système ou programmes créés par un utilisateur
- Exemple: `gcc -o test test.c ; ./test`
  Produit programme exécutable dans fichier `test`; exécute le programme test
- Question: pourquoi `./test` ?

Fichiers de données

- Documents, images, programmes sources, etc.
- Convention: le suffixe indique la nature du contenu
  Exemples : .c (programme C), .o (binaire translatable, cf. plus loin), .h (entête C), .gif (un format d'images), .pdf (Portable Document Format), *etc.*
  Remarque: ne pas faire une confiance aveugle à l'extension (*cf.* `man file`)

Fichiers temporaires servant pour la communication

- Ne pas oublier de les supprimer après usage
- On peut aussi utiliser des tubes (*cf.* RS l'an prochain)

Protection des fichiers: généralités

Définition (générale) de la sécurité

- confidentialité :
- intégrité :
- contrôle d'accès :
- authentification :

Comment assurer la sécurité

- Définition d'un ensemble de règles (politiques de sécurité) spécifiant la sécurité d'une organisation ou d'une installation informatique
- Mise en place mécanismes de protection pour faire respecter ces règles

Règles d'éthique

- Protéger ses informations confidentielles (comme les projets et TP notés!)
- Ne pas tenter de contourner les mécanismes de protection (c'est la loi)
- Règles de bon usage avant tout:
  La possibilité technique de lire un fichier ne donne pas le droit de le faire

Protection des fichiers: généralités

Définition (générale) de la sécurité

- confidentialité : informations accessibles aux seuls usagers autorisés
- intégrité :
- contrôle d'accès :
- authentification :

Comment assurer la sécurité

- Définition d'un ensemble de règles (politiques de sécurité) spécifiant la sécurité d'une organisation ou d'une installation informatique
- Mise en place mécanismes de protection pour faire respecter ces règles

Règles d'éthique

- Protéger ses informations confidentielles (comme les projets et TP notés!)
- Ne pas tenter de contourner les mécanismes de protection (c'est la loi)
- Règles de bon usage avant tout:
  La possibilité technique de lire un fichier ne donne pas le droit de le faire

Protection des fichiers: généralités

Définition (générale) de la sécurité

- confidentialité : informations accessibles aux seuls usagers autorisés
- intégrité : pas de modifications indésirées
- contrôle d'accès :
- authentification :

Comment assurer la sécurité

- Définition d'un ensemble de règles (politiques de sécurité) spécifiant la sécurité d'une organisation ou d'une installation informatique
- Mise en place mécanismes de protection pour faire respecter ces règles

Règles d'éthique

- Protéger ses informations confidentielles (comme les projets et TP notés!)
- Ne pas tenter de contourner les mécanismes de protection (c'est la loi)
- Règles de bon usage avant tout:
  La possibilité technique de lire un fichier ne donne pas le droit de le faire

Protection des fichiers: généralités

Définition (générale) de la sécurité

- confidentialité : informations accessibles aux seuls usagers autorisés
- intégrité : pas de modifications indésirées
- contrôle d'accès : qui a le droit de faire quoi
- authentification :

Comment assurer la sécurité

- Définition d'un ensemble de règles (politiques de sécurité) spécifiant la sécurité d'une organisation ou d'une installation informatique
- Mise en place mécanismes de protection pour faire respecter ces règles

Règles d'éthique

- Protéger ses informations confidentielles (comme les projets et TP notés!)
- Ne pas tenter de contourner les mécanismes de protection (c'est la loi)
- Règles de bon usage avant tout:
  La possibilité technique de lire un fichier ne donne pas le droit de le faire

Protection des fichiers: généralités

Définition (générale) de la sécurité

- confidentialité : informations accessibles aux seuls usagers autorisés
- intégrité : pas de modifications indésirées
- contrôle d'accès : qui a le droit de faire quoi
- authentification : garantie qu'un usager est bien celui qu'il prétend être

Comment assurer la sécurité

- Définition d'un ensemble de règles (politiques de sécurité) spécifiant la sécurité d'une organisation ou d'une installation informatique
- Mise en place mécanismes de protection pour faire respecter ces règles

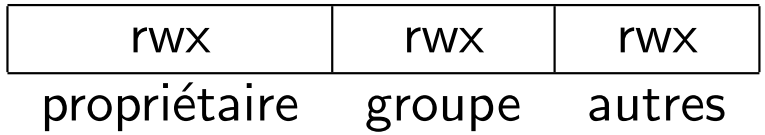
Règles d'éthique

- Protéger ses informations confidentielles (comme les projets et TP notés!)
- Ne pas tenter de contourner les mécanismes de protection (c'est la loi)
- Règles de bon usage avant tout:
  La possibilité technique de lire un fichier ne donne pas le droit de le faire

Protection des fichiers sous Unix

Sécurité des fichiers dans Unix

- Trois types d'opérations sur les fichiers : lire (r), écrire (w), exécuter (x)
- Trois classes d'utilisateurs vis à vis d'un fichier:
  propriétaire du fichier ; membres de son groupe ; les autres



  Granularité plus fine avec les Access Control List (peu répandus, pas étudiés ici)
- Pour les répertoires, `r` = `ls`, `w` = créer des éléments et `x` = `cd`.
- `ls -l` pour consulter les droits; `chmod` pour les modifier (cf. `man chmod`)

Mécanisme de délégation

- Problème : programme dont l'exécution nécessite des droits que n'ont pas les usagers potentiels (exemple: gestionnaire d'impression, d'affichage)
- Solution (setuid ou setgid): ce programme s'exécute toujours sous l'identité du propriétaire du fichier; identité utilisateur momentanément modifiée identité réelle (celle de départ) vs identité effective (celle après setuid)

Crash course on using the terminal

Main idea

- Your shell is somewhere in the filesystem tree (current directory)
- You issue commands to interact with the system

Commands Basic Syntax

- Every command follows this syntax: `<command name> <arguments>`
- Arguments are space separated
- Flags are specific arguments begining usually with - (minus)

Minimal set of commands to remember

Action	**Command**	**Memoing**
Examine content of current dir	ls	listing
Know name of current dir	pwd	Print Working Directory
Change current dir	cd	change directory
Copy a file into another Create a new dir	cp mkdir	copy make directory
Destroy a file, a dir	rm, rmdir	remove
Usual shorthand for files and dirs	. .. / ~ * ~user	

Using the terminal efficiently

Common Tricks

- Typing everything is really to slow. You need to be lazy here.
- Up / Down : see commands typed previously. Edit it, and go!
- Ctrl-A / Ctrl-E : jump to begin/end of line
- Tab : auto-complete what you are currently typing

Medium Tricks

- Ctrl-R : begin to search a text pattern in the command history
- `!command` : directly relaunch the last command involving that `command`
- `!!` : directly relaunch the last command

Advanced Tricks

- Master your terminal (know the base commands)
- Assemble commands in pipe to get more advanced ones
- Write one-line scripts directly in the terminal
- Configure your environment: Declare aliases, write scripts, etc.

Conclusion on Unix (for now)

Unix is one of the most influent operating system

- Around since 40 years, still there for a long time
- Most of the OS research inovation go in Unix first (open source power)
- Other OSes become Unixes (OS X) or get portability layers (z/OS, windows)

You can use that powerful tool too

- Not as much game as on your Wii, but fully usable and free
- The interface may be different of what you're used to
- May be less intuitive at first glance, but there's a strong underlying philosophy
- Constitute a playground of choice for CS students

Mastering this system is the goal of that course

Chapter 2

C as Second Language

- Syntax of the C language
  - C Quick Reference
  - Type Constructors
  - Lexical Structure
- Interactions with the Environment
  - Input/Output: Terminal and Files
  - Command-line Arguments
  - Interacting with Processes
- Associated Tools
  - Preprocessor

We said that C and Java are quite similar

Similarities between C and Java

- Operators
  - Arithmetic (+,-,*,/,% ; ++,- -,*=,...); Bitwise (&,|,^,!,<<,>>)
  - Relational (<,>,<=,>=,==,!=); Logical &&, ||, !, (a?b:c)
- Keywords and Language Constructs
  - `if( ){ } else { }`
  - `while( ){ }`
  - `switch( ) { case 0: ... }`
  - `for(i=0; i<100; i++){ }`
  - `do { } while( )`
  - `break`, `continue`, `return`
- Basic (primitive) types: void, int, short, long; float, double; char.
  No boolean, use int instead (0=False; anything else=True)
- Function declarations: `int fact(int a){return a==0 ? 1 : a*fact(a-1);}`

Differences between C and Java

- No exception: usually rely on int error code instead (and usually a pain)
- No class/package/interface: code modularity different (not compiler-enforced)
- No garbage collector: alloc and free manually needed memory (incredible pain)
- Terse standard library: reimplement your datastructures (but tons of extra libs)

Paradigm difference between C and Java

The syntax is not everything. Java and C are really different

Paradigm shift seen from the C side

- Object-Oriented Programming Paradigm
  - Decide which classes you need
  - Provide a full set of operations for each class
  - Make commonality explicit by using inheritance
- Procedural Programming Paradigm
  - Decide which procedures and data structures you want
  - Use the best algorithms

Paradigm difference between C and Java

The syntax is not everything. Java and C are really different

Paradigm shift seen from the C side

- Object-Oriented Programming Paradigm
  - Decide which classes you need
  - Provide a full set of operations for each class
  - Make commonality explicit by using inheritance
- Procedural Programming Paradigm
  - Decide which procedures and data structures you want
  - Use the best algorithms

Reality is a bit different

- Nothing forces you to any sort of organization in C. You're free of the worst

  Oh, I am a C programmer and I'm okay.
  I muck with indices and structs all day.
  And when it works, I shout hoo-ray.
  Oh, I am a C programmer and I'm okay.
- (but you're free of the best, too, even if "good style in C" is a relative notion)

C Quick Reference

We won't get into details here. References are for later use, not for beginners

Complete list of keywords (in ANSI C)

- Storage specifiers: `auto register static extern typedef`
- Type specifiers: `char double enum float int long short signed struct union unsigned void` (+`sizeof`, which is an operator on types)
- Type quantifiers: `const volatile`
- Controls: `break case continue default do else for goto if return switch while`

Operators Precedence (and Associativity)

1. Functions calls, subscripting and selection: () [] -> . →
2. Not: ! ~ Inc/Dec: ++ −− Unary - Cast (type) Indir./address * & sizeof ←
3. Math operators: * / %
4. Other math operators: + -(binary) →
5. Bitwise shifts: << >>
6. Relational operators: < <= > >= →
7. Equality: == !=
8. Bitwise AND: &
9. Bw XOR: ^
10. Bw OR: | →
11. Logical AND: &&
12. Logical OR: || →
13. Ternary Operator ?: (condition ? exprIfTrue : exprIfFalse) ←
14. Assignments with operator: = += -= *= /= %= &= ^= |= <<= >>= ←
15. Sequencing expressions: , (comma) →

C base types

C and the types

- The C language is (really) weakly typed (wrt CAML for example)
- C types look like Java ones at the first glance, but include some . . . surprises

What defines a type in computer languages?

- Value domain: what can be encoded in that type
- Operators: what can be done with values of that type

C base types

C and the types

- The C language is (really) weakly typed (wrt CAML for example)
- C types look like Java ones at the first glance, but include some . . . surprises

What defines a type in computer languages?

- Value domain: what can be encoded in that type
- Operators: what can be done with values of that type

Existing types in the C language

- **void:** Domain: $\emptyset$ (none); Operators: none
  Placeholder for type where there is no value (type of return when no return)
- **int:** Domain: integers; Operators: All numerical, logical and bitwise ones
  Variants: short/long and also signed/unsigned
- **float** and **double:** floating point numbers (IEEE754 compliant, no variant)
- **char:** Domain: chars such as 'a', '1', '$' and some less common ones
  Operators: numerical, logical and bitwise ones. Variants: signed/unsigned
  Yep, chars are "small numbers" in C

Beware, type sizes are not known in C

Type	Java	C	
char	16 bits	8 bits	
short	16 bits	16 bits	
int	32 bits	16, 32 or 64 bits	("most natural size for architecture")
long	64 bits	32 or 64 bits	
float	32 bits	32 bits	
double	64 bits	64 bits	
boolean	1 bit	–	No such thing in C, use int (or bit fields)
byte	8 bits	–	Doesn't exist, use char
long long	–	64 bits	This type is not standard/unofficial
long double	–	80, 96 or 128 bits	this one either

Type domains also naturally vary

Type size	Range when signed		Range when unsigned	
8 bits	$[-2^7; 2^7[$ =	$[-128; 128[$	$[0; 2^8[$ =	$[0; 256]$
16 bits	$[-2^{15}; 2^{15}[$ =	$[-32\,768; 32\,768[$	$[0; 2^{16}[$ =	$[0; 65\,535]$
32 bits	$[-2^{31}; 2^{31}[$ =	$[-2\,147\,483\,647; 2\,147\,483\,648]$	$[0; 2^{32}[$ =	$[0; 4\,294\,967\,295]$
64 bits	$[-9\,223\,372\,036\,854\,775\,807; 9\,223\,372\,036\,854\,775\,808]$		$[0; 18\,446\,744\,073\,709\,551\,615]$	

Use `sizeof()` when you need to know a type size on current machine
(and the limits.h file)

Chapter 2

C as Second Language

- Syntax of the C language
  - C Quick Reference
  - Type Constructors
  - Lexical Structure
- Interactions with the Environment
  - Input/Output: Terminal and Files
  - Command-line Arguments
  - Interacting with Processes
- Associated Tools
  - Preprocessor

Type Constructors

How to keep together related data grouped in C?

- Arrays (similar to Java) ordered list of elements
  Ex: Values of the fibonacci suite; Temperature over the time; Data to sort ...
- Structures: like java objects without methods, or SQL reccords
  Ex: A car; A student; A group of students; A school ...
- Enumerations group of related values (exists also in Java, but rarely used)
  Ex: Colors; Cards in a deck; Direction (north/south/east/west)...
- Unions: Like structures, but stores everything at the same memory location
  Advanced stuff, useful for strange memory tricks (data conversion)
- Bit fields: arrays of bits. Advanced stuff allowing direct access to memory
  Useful to encode several booleans in a compact way

Let's detail the basic ones

- Aka, Arrays, structures and enumerations.
- Unions and bit fields are kinda advanced C-fu

Arrays in C

Similarity to Java

- Defining: `int T[5]` defines 5 integers, noted T[0], T[1], T[2], T[3] and T[4]
- Initialization: `int T[5] = {10,20,30,40,50};` does what you expect
  For the record, in Java, you'd write `int[] T = new int[] {10,20,30,40,50};`
- No global operators: `Ta==Tb` and `Ta+Tb` ... does not do what you think

C arrays specificities from Java ones

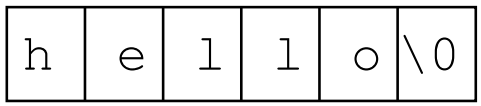
- You must write `int T[5]` because `int[5] T` is forbidden
  To understand a C (or Java) complex type, you must read from right to left
- You cannot retrieve the size of an array: `T.length()` does not exist
  You must store the array size alongside to the array, in an integer
- Dynamically sized arrays are not allowed in C [without dynamic memory]
  - Array sizes must be known at compilation time
  - `int T[] = new int[a];` is just impossible (in ANSI C)
- There is no bound checking on arrays in C (and C memory is a big magma)

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:

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



- 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$\rightsquigarrow$yellow)

Java enums

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

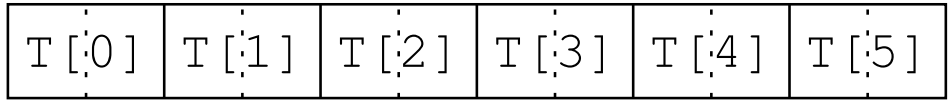
Memory layout of C type constructors

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

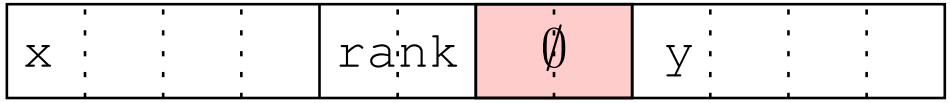
```
struct point {
  double x;
  double y;
  int rank;    };
```

```
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)

Chapter 2

C as Second Language

- Syntax of the C language
  - C Quick Reference
  - Type Constructors
  - Lexical Structure
- Interactions with the Environment
  - Input/Output: Terminal and Files
  - Command-line Arguments
  - Interacting with Processes
- Associated Tools
  - Preprocessor

Lexical Structure of a Typical Program

- Header inclusions: Load the prototypes of function that you want to use
  - Lines begin with `#include`
  - Loaded files are called **headers**
  - `#include <file>` for system-wide headers
  - `#include "file"` for project-wide headers

```
#include <stdlib.h>
#include <stdio.h>
#include "my_prototypes.h"
```

- Preprocessor directives; types defs; globals&constants; function prototypes
  - `typedefs` as seen before
  - `const` are just like final in Java
  - Globals visible from the whole program
  - Prototypes tell the compiler about functions

```
#define MAX 42
const int size = 5;
int ranking = 3;
int array[MAX];
void compute_ranking(int from);
```

- Function definitions, including the `main()` function
  - There must be one unique `main()` function
  - Program entry point: started first
  - *Should* return `EXIT_SUCCESS` or `EXIT_FAILURE`
  - Several prototypes exist, this one is classical

```
void main(int argc,char *argv[]){
  /* My code here */
  return EXIT_SUCCESS;
}
```

- The program can spread over several files (more to come on this)

Source Formatting Best Practices

Identifier naming schema

- There is a religion war between `this_naming_schema` and `thisNamingSchema`
- I personally use the first one in C, and the second one in Java
- Pick your own, and STICK TO IT!

Indenting

- There is another religion war between these two styles (and others)

```
if (cond) {
  /* body */
}
```

```
if (cond)
  {
    /* body */
  }
```

- I personally use the first one (more compact), but YMMV:
- As long as you DO indent your code consistently, that's fine with me

Spacing (no real flame war here, boring)

- No spaces around these: `->` `.` `[]` `!` `~` `++` `--` `&` unary `*` and `-`
- One space around these: `=` `+=` `?:` `+` `<` `&&` and binary operators

Chapter 2

C as Second Language

- Syntax of the C language
  - C Quick Reference
  - Type Constructors
  - Lexical Structure
- Interactions with the Environment
  - Input/Output: Terminal and Files
  - Command-line Arguments
  - Interacting with Processes
- Associated Tools
  - Preprocessor

Terminal I/O

Interactions with the external world in C and Java

- Java: easy to build a GUI and painful to interact through the console
- C: the contrary (GUI require external libs such as Gnome, KDE, ncurse)

Standard Communication Channels

- Standard input (stdin): keyboard, unless it got redirected
- Standard output (stdout): screen, unless it got redirected
- Standard error (stderr): screen, unless it got redirected
- Example of redirection: `prog < in_file > out_file 2>err_file`

Single character I/O

- `int getchar()`: returns the next character from input
  (or EOF in case of End Of File, this constant is defined in stdio.h)
- `int putchar(int c)`: writes c to output
- Yes these function consider chars as ints. Sorry.

Multiple Characters Terminal I/O

Motivation

- Single char I/O works, but that's a real pain. We want the equivalent of

```
System.out.println("hello "+name+".  How are you today?");
```

- No `tostring()` magic functions nor magic `+` string concatenation in C

Interacting with the terminal in C

- Actually there is two major interfaces for that in C
- Low-level API (`write()` / `read()`): better performance *when used correctly*
- High-level API (`printf()` / `scanf()`): easier to use; way to go this year
- You need to load `stdio.h` to use all these functions

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

Chapter 2

C as Second Language

- Syntax of the C language
  - C Quick Reference
  - Type Constructors
  - Lexical Structure
- Interactions with the Environment
  - Input/Output: Terminal and Files
  - Command-line Arguments
  - Interacting with Processes
- Associated Tools
  - Preprocessor

Command line arguments

Motivation

- Classical tools such as `ls` or `mv` get arguments from the command line
- How can we do the same? From the `main()` arguments of course

Command line arguments

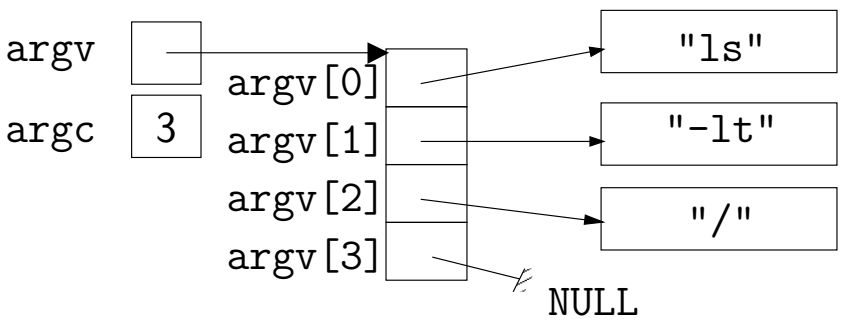
Motivation

- Classical tools such as `ls` or `mv` get arguments from the command line
- How can we do the same? From the `main()` arguments of course

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

- `argc`: amount of parameter received; `argv`: array of strings received
- (note: these names are conventions, doing really otherwise hinders readability)

Memory layout for `ls -lt /`



Command line arguments

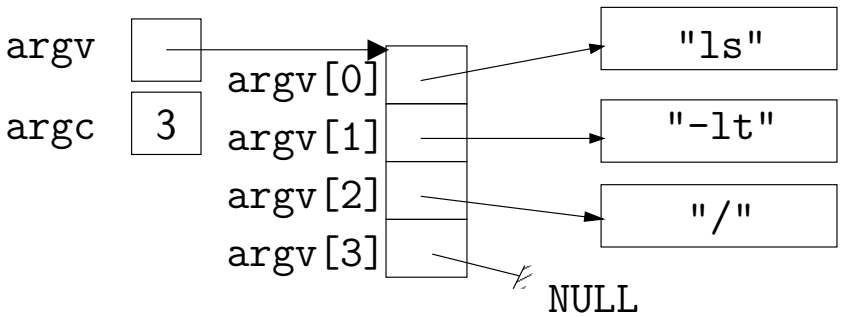
Motivation

- Classical tools such as `ls` or `mv` get arguments from the command line
- How can we do the same? From the `main()` arguments of course

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

- `argc`: amount of parameter received; `argv`: array of strings received
- (note: these names are conventions, doing really otherwise hinders readability)

Memory layout for `ls -lt /`



Displaying the arguments

```
int main(int argc, char *argv[]) {
  int i;
  for (i=1; i<argc; i++) {
    printf("Argument #%d:  %s\n",
           i, argv[i]);
  }
  return EXIT_SUCCESS;
}
```

Chapter 2

C as Second Language

- Syntax of the C language
  - C Quick Reference
  - Type Constructors
  - Lexical Structure
- Interactions with the Environment
  - Input/Output: Terminal and Files
  - Command-line Arguments
  - Interacting with Processes
- Associated Tools
  - Preprocessor

Interacting with Processes

First of all, what is a process?

Interacting with Processes

First of all, what is a process?

- That's a box encapsulating the execution of a task
- The operating system uses these boxes to let several tasks coexist in memory
- Processes are to programs what objects are to classes: living instances
  You can use the same program than me, but you cannot use my processes

Interacting with Processes

First of all, what is a process?

- That's a box encapsulating the execution of a task
- The operating system uses these boxes to let several tasks coexist in memory
- Processes are to programs what objects are to classes: living instances
  You can use the same program than me, but you cannot use my processes

Basic shell interaction

- Start a process, simply type the name of the program with arguments
  With &, the process runs in background. Ex: `emacs &`
  Else, CTRL-Z suspends process; then `bg` ⇝ background; `fg` ⇝ foreground
- List all existing processes `ps -ef` all mine `ps -aux` bob's `ps -u bob`
- Kill a process knowing its PID: `kill` *pid*
- Kill a process knowing its name: `killall` *name*

Interacting with Processes from C

Starting an external process

- This is as easy as `system("mkdir /tmp/directory")`
- Trick 1: the return value is a bit counter-intuitive (0 –false– if ok)
- Trick 2: stdin/stdout of started process get to stdin/stdout of father
  This limits the possible interaction between both processes

Interacting with Processes from C

Starting an external process

- This is as easy as `system("mkdir /tmp/directory")`
- Trick 1: the return value is a bit counter-intuitive (0 –false– if ok)
- Trick 2: stdin/stdout of started process get to stdin/stdout of father
  This limits the possible interaction between both processes

Starting and interacting with external processes

- Use `FILE* popen(char *command, char *type)` for that
- If type is "r", read from process. If "w", write to it (cannot do both this way)
- Use `pclose(FILE*handle)` instead of `fclose()` to close such a descriptor
- After the RS course, you'll find implementing `popen` boring because simple

Chapter 2

C as Second Language

- Syntax of the C language
  - C Quick Reference
  - Type Constructors
  - Lexical Structure
- Interactions with the Environment
  - Input/Output: Terminal and Files
  - Command-line Arguments
  - Interacting with Processes
- Associated Tools
  - Preprocessor

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 $\rightsquigarrow$ 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` ⇝ 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 ⇝ 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 "biiiiirk"`

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

(this ends the second lecture)

Chapter 2

C as Second Language

- Syntax of the C language
  - C Quick Reference
  - Type Constructors
  - Lexical Structure
- Interactions with the Environment
  - Input/Output: Terminal and Files
  - Command-line Arguments
  - Interacting with Processes
- Associated Tools
  - Preprocessor

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

Memory Management in C

Introduction

- Main specificity of the C language: **Memory Management**
- You have **full control** over the memory in C
- That gives you the full power ... to shoot you in the foot

Lecture agenda

- First explore the basic notions over memory
  - Local and Global variables; Scope and Lifetime; Notion of Address and Pointers
- Then, (quick) look at the system side of memory management
  - Memory Layout of a typical UNIX Process (more details in RS next year)
- Finally, go into the full details of memory allocation/deallocation
  - Student's hated malloc and associated madness

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

```
 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?

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* ☐

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 [0]

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?

I1 a 0

I3 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* 0

l3 *b* 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* 0

l3 *b* 0

l5 `a: 0; b: 0`

l6 *a* 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 0

l3 b_3 0

l5 `a: 0; b: 0`

l6 a_1 5

l8 a_8 ??

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 0

l3 b_3 0

l5 `a: 0; b: 0`

l6 a_1 5

l8 a_8 ??

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 0

l3 b_3 0

l5 `a: 0; b: 0`

l6 a_1 5

l8 a_8 ??

l9 `a: ??; b: 0`

l11 a_1 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 0

l3 b_3 0

l5 `a: 0; b: 0`

l6 a_1 5

l8 a_8 ??

l9 `a: ??; b: 0`

l11 a_1 10

l13 b_{13} 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 0

l3 b_3 0

l5 `a: 0; b: 0`

l6 a_1 5

l8 a_8 ??

l9 `a: ??; b: 0`

l11 a_1 10

l13 b_{13} 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 0

l3 b_3 0

l5 `a: 0; b: 0`

l6 a_1 5

l8 a_8 ??

l9 `a: ??; b: 0`

l11 a_1 10

l13 b_{13} 10

l14 `a: 10; b: 10`

l15 b_{13} 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 0

l3 b_3 0

l5 `a: 0; b: 0`

l6 a_1 5

l8 a_8 ??

l9 `a: ??; b: 0`

l11 a_1 10

l13 b_{13} 10

l14 `a: 10; b: 10`

l15 b_{13} 15

l17 b_{17} 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 0

l3 b_3 0

l5 `a: 0; b: 0`

l6 a_1 5

l8 a_8 ??

l9 `a: ??; b: 0`

l11 a_1 10

l13 b_{13} 10

l14 `a: 10; b: 10`

l15 b_{13} 15

l17 b_{17} 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 0

l3 b_3 0

l5 a: 0; b: 0

l6 a_1 5

l8 a_8 ??

l9 a: ??; b: 0

l11 a_1 10

l13 b_{13} 10

l14 a: 10; b: 10

l15 b_{13} 15

l17 b_{17} 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 0

l3 b_3 0

l5 a: 0; b: 0

l6 a_1 5

l8 a_8 ??

l9 a: ??; b: 0

l11 a_1 10

l13 b_{13} 10

l14 a: 10; b: 10

l15 b_{13} 15

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

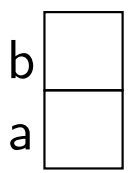
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

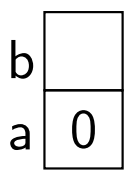
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

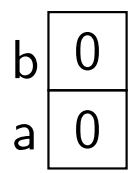
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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



Variables are Stored on a Stack

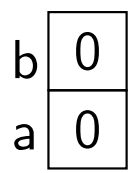
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

Variables are Stored on a Stack

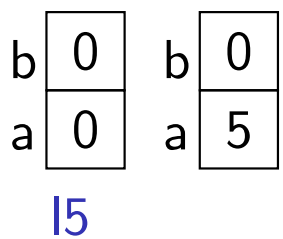
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

Variables are Stored on a Stack

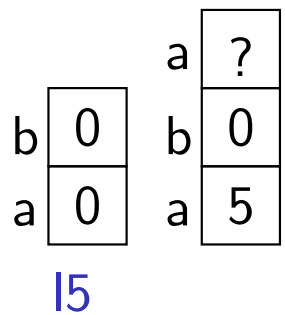
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

Variables are Stored on a Stack

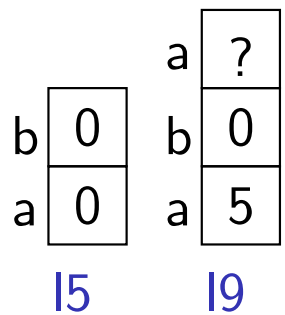
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

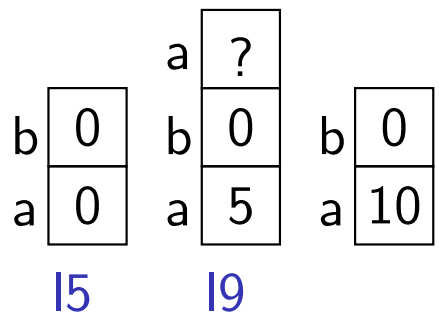
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

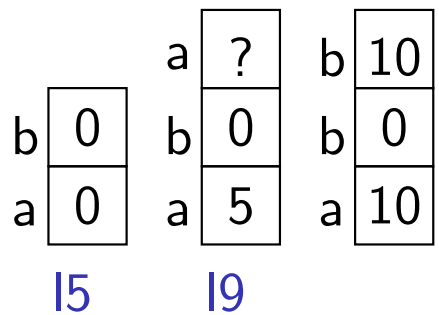
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

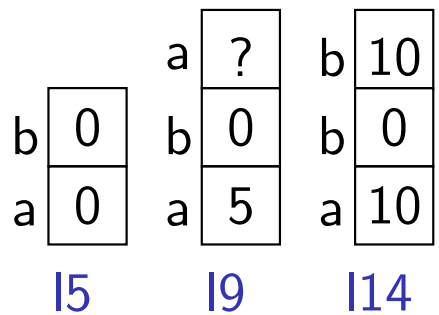
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

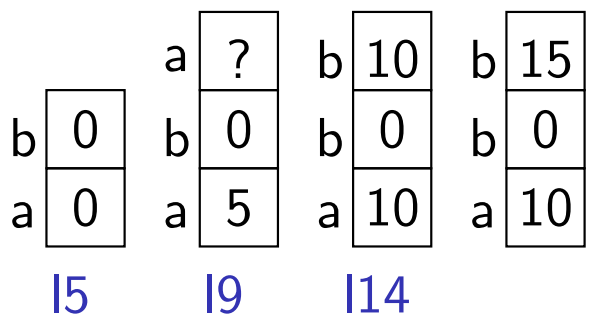
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

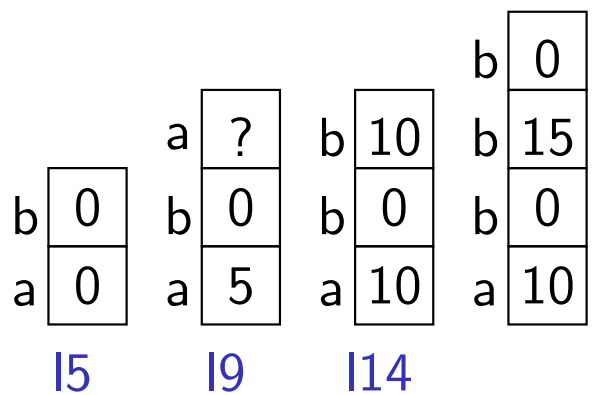
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

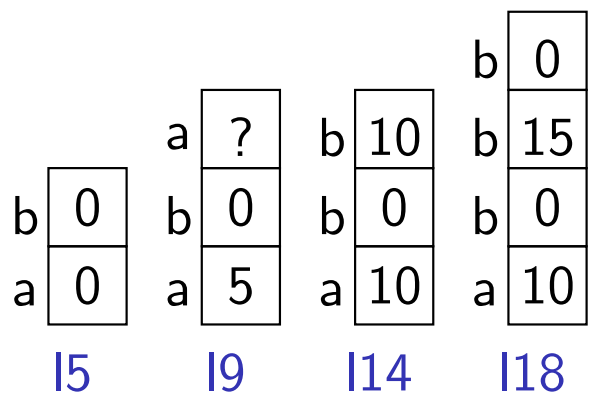
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

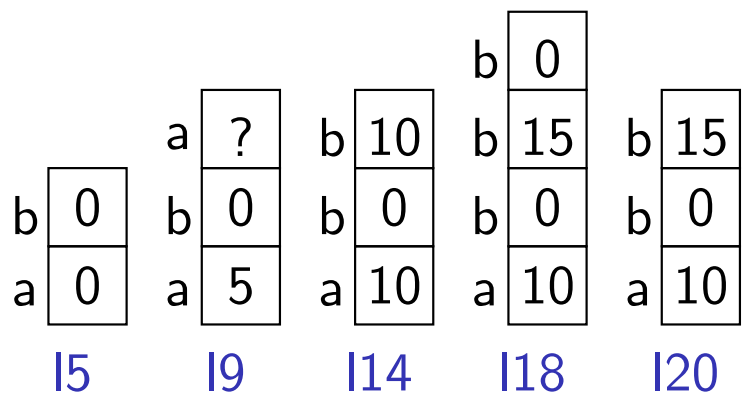
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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

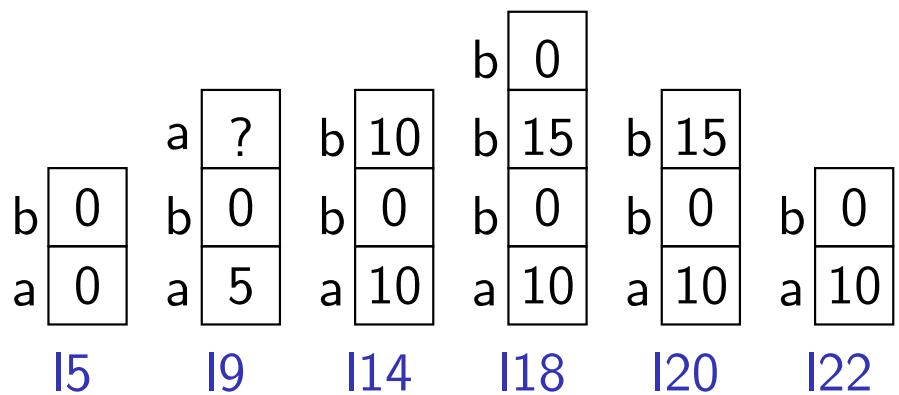
```
 1:int a;
 2:int main() {
 3:   int b;
 3:   a=0;
 4:   b=a;
 5:   printf("a: %d, b: %d\n",a,b);
 6:   a += 5;
 7:   {
 8:     int a;
 9:     printf("a: %d, b: %d\n",a,b);
10:   }
11:   a += 5;
12:   {
13:     int b=a;
14:     printf("a: %d, b: %d\n",a,b);
15:     b += 5;
16:     {
17:       int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



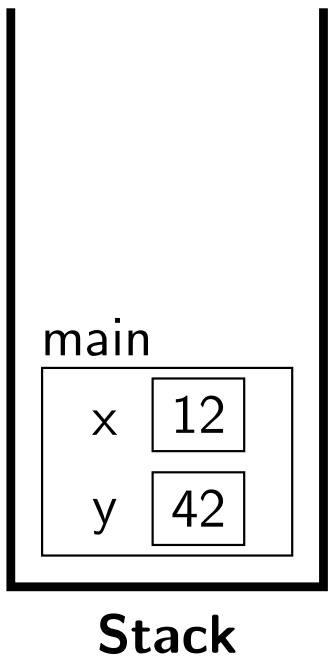
Function Parameters

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);
}
```



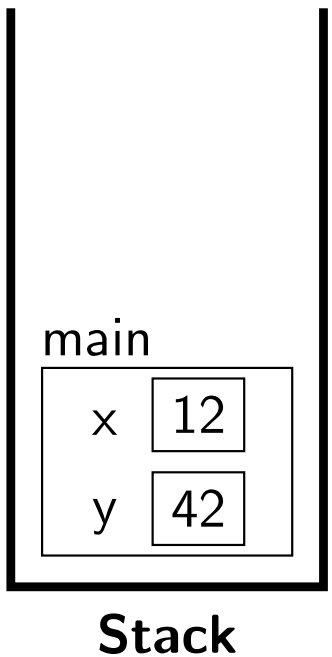
Function Parameters

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);
}
```



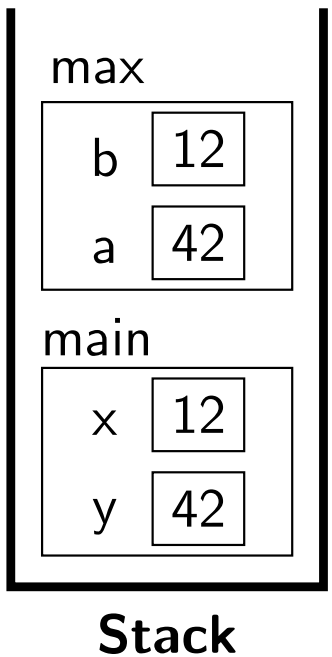
Function Parameters

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);
}
```



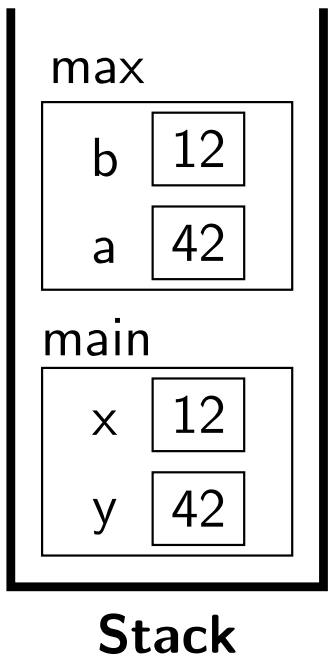
Function Parameters

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);
}
```



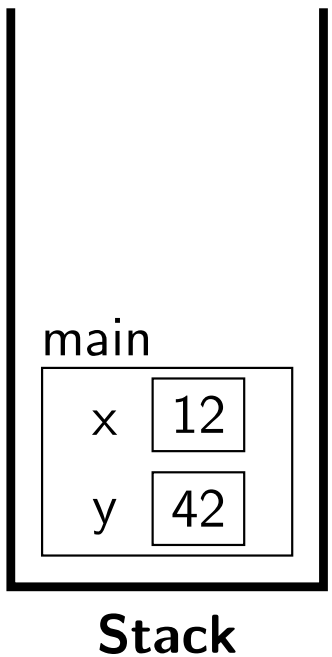
Function Parameters

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)

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

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



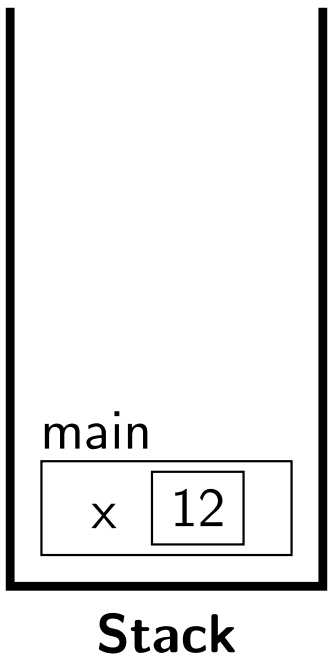
Function Parameters Tricks

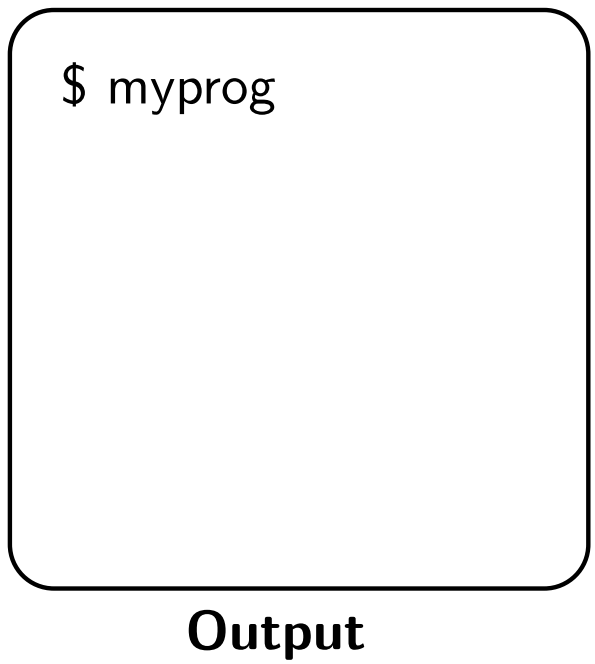
Parameters are passed by value

- 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;
}
```





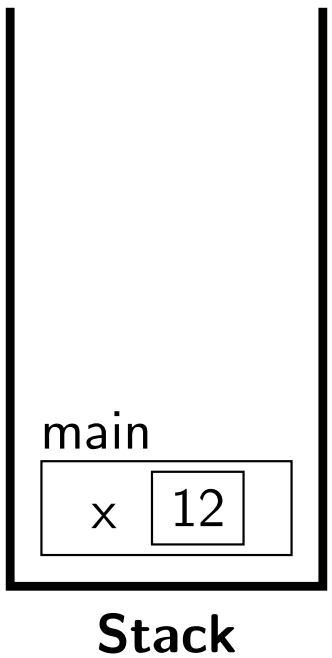
Function Parameters Tricks

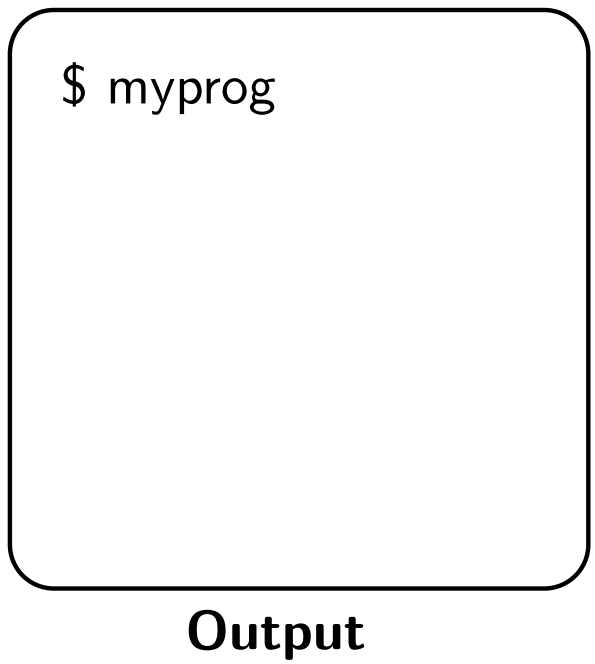
Parameters are passed by value

- 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;
}
```





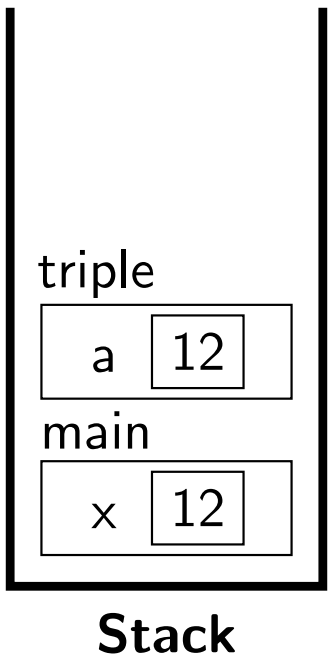
Function Parameters Tricks

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

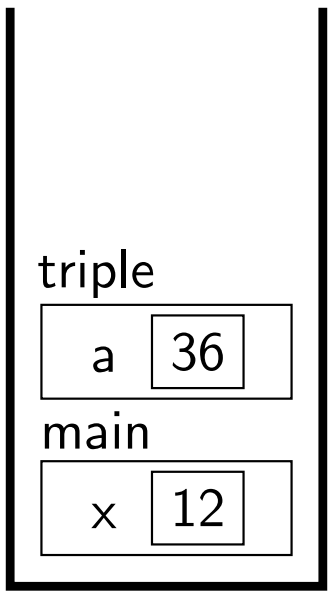
Function Parameters Tricks

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

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

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



Stack

Output

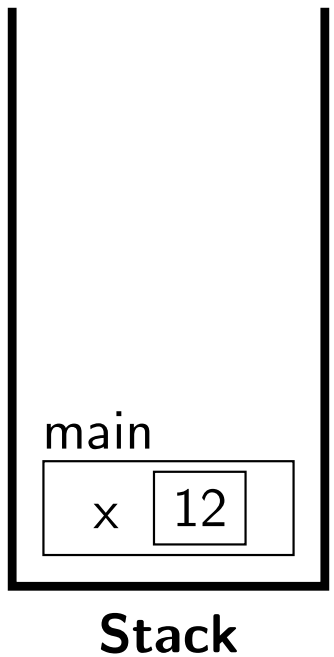
Function Parameters Tricks

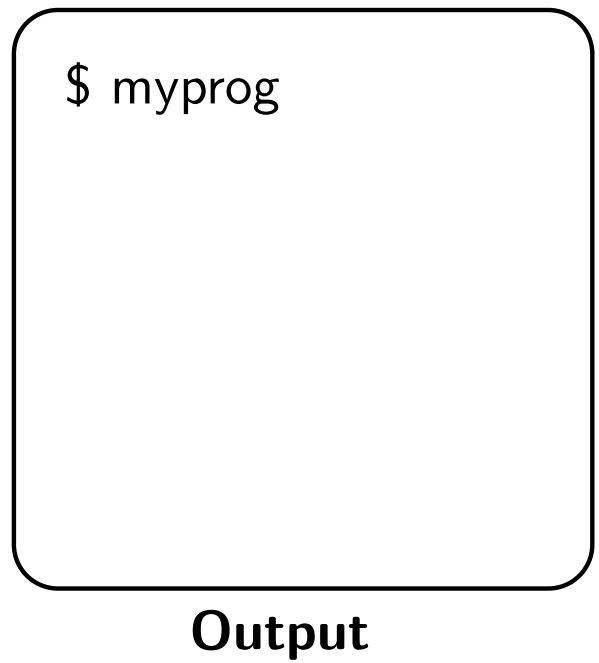
Parameters are passed by value

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

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

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





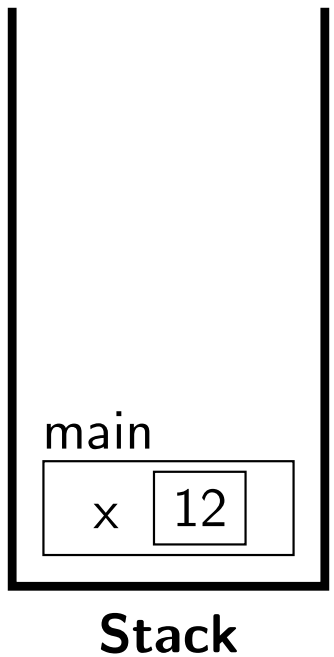
Function Parameters Tricks

Parameters are passed by value

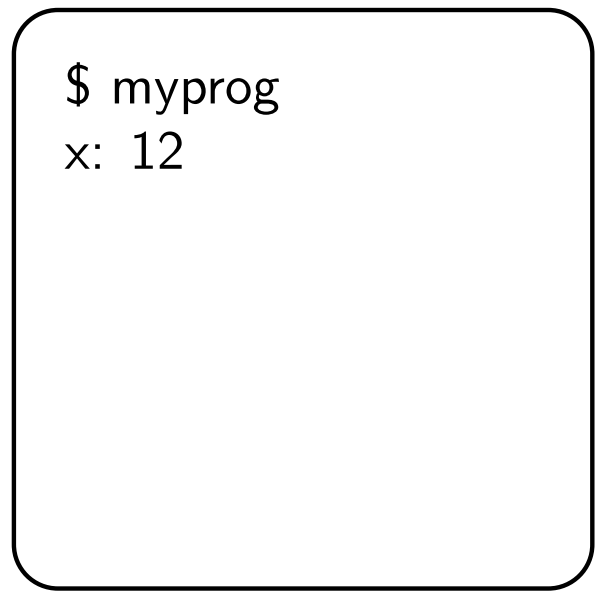
- We just said that but it is not as natural as it seems
- It forbids any side effects on parameters

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

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



Stack



Output

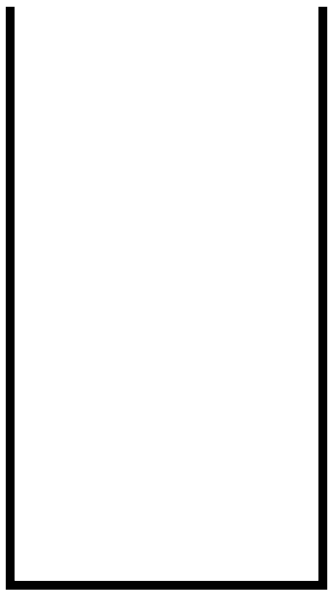
Function Parameters Tricks

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
- There is no way to avoid passing by value
- But pointers help: scanf manages to "modify its arguments"

```
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

```
 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()

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

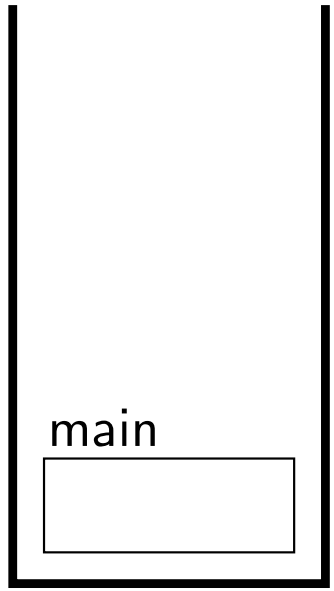
- Increases lifetime: from "for this call" to "for ever" (as if it were global)
- Visibility remains unchanged
- Similar concept in Java: static

	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

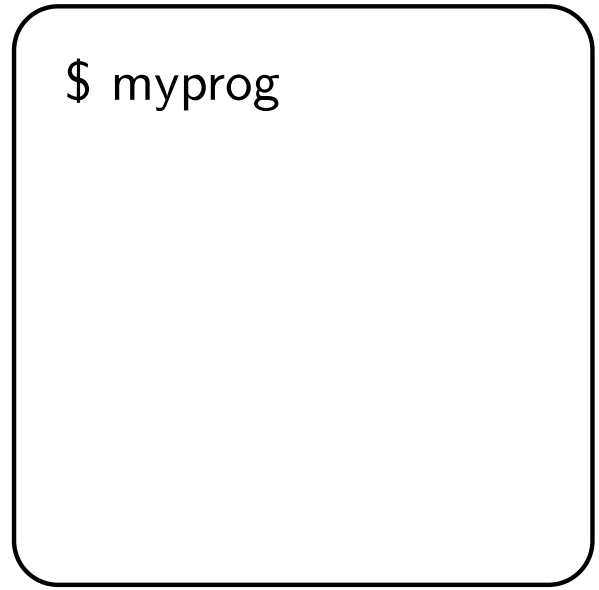
More on Static Local Variables

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

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



Stack

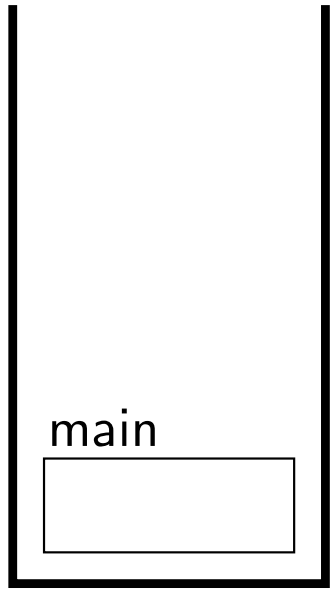


Output

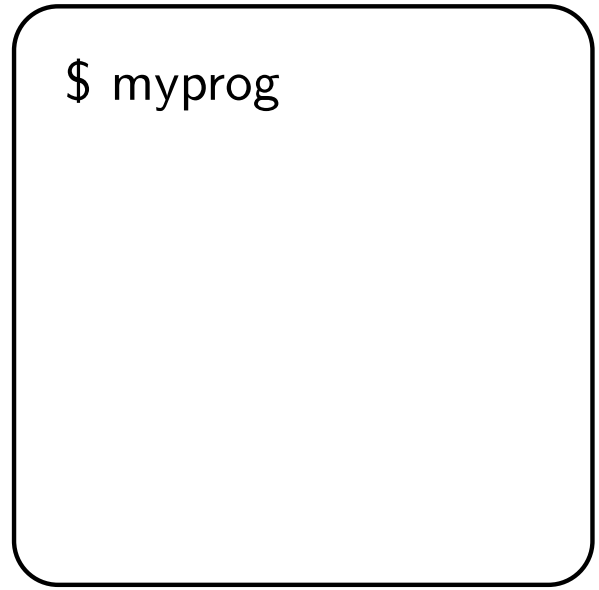
More on Static Local Variables

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

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



Stack

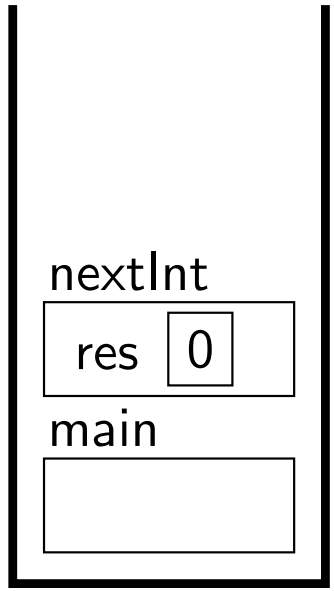


Output

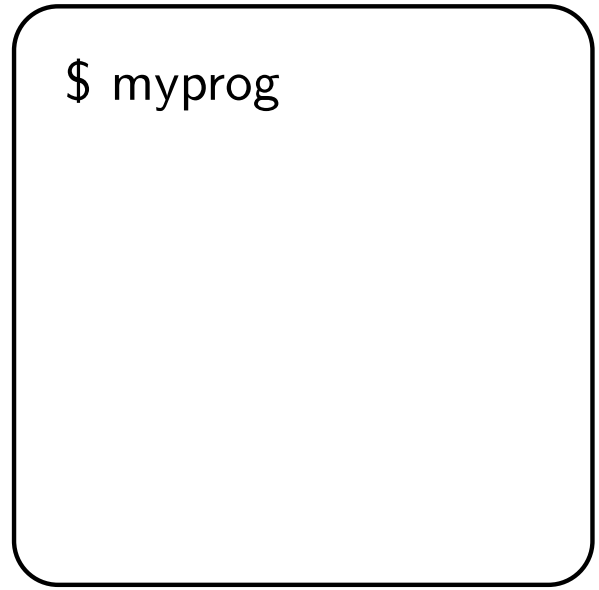
More on Static Local Variables

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

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



Stack

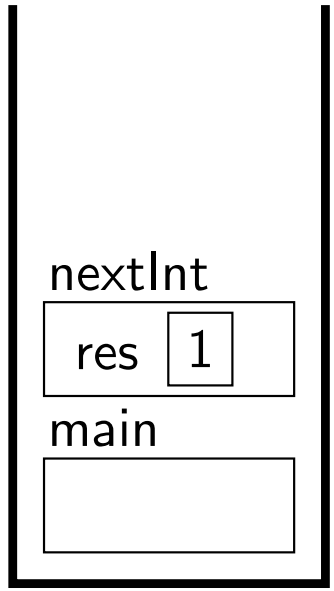


Output

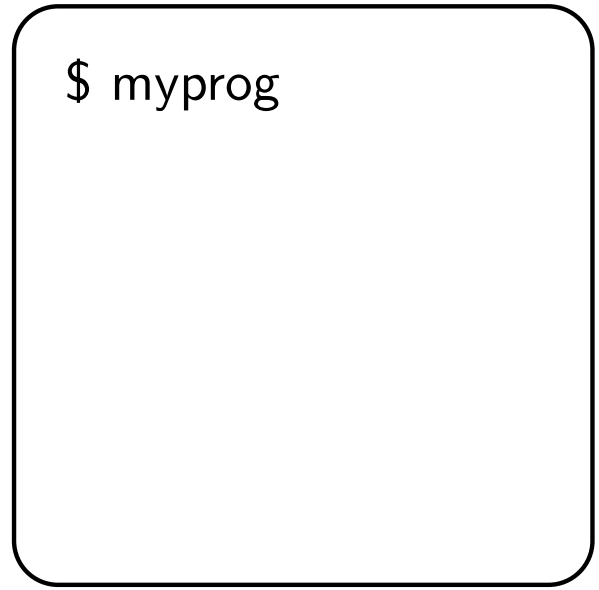
More on Static Local Variables

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

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



Stack

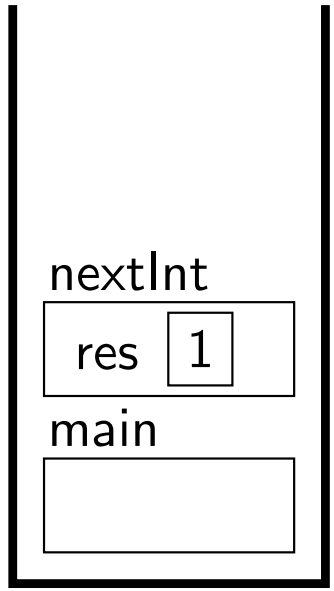


Output

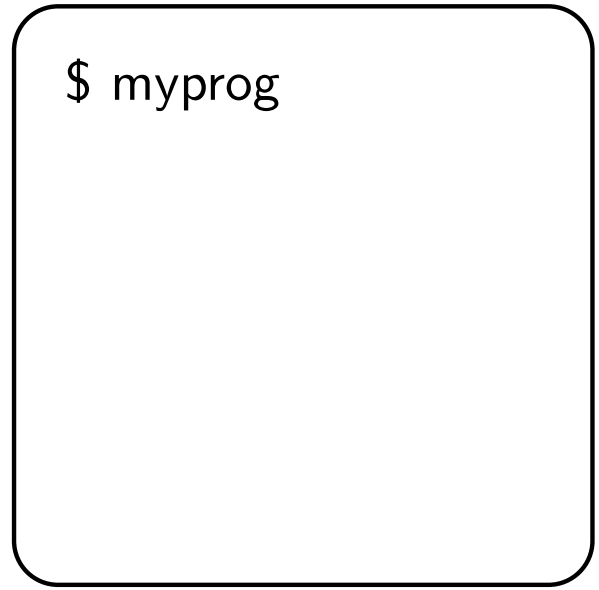
More on Static Local Variables

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

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



Stack

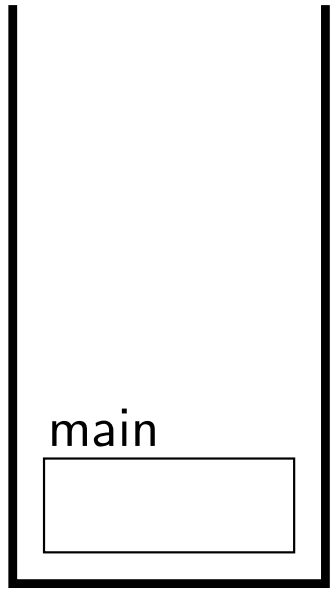


Output

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

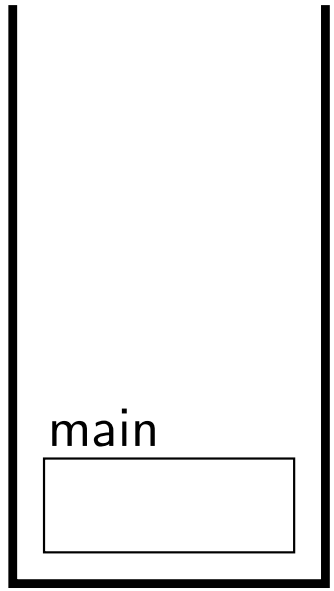
```
$ myprog
next: 1
```

Output

More on Static Local Variables

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

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



Stack

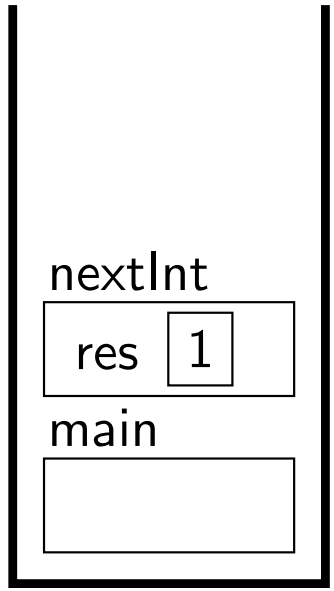
```
$ myprog
next: 1
```

Output

More on Static Local Variables

```
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
```

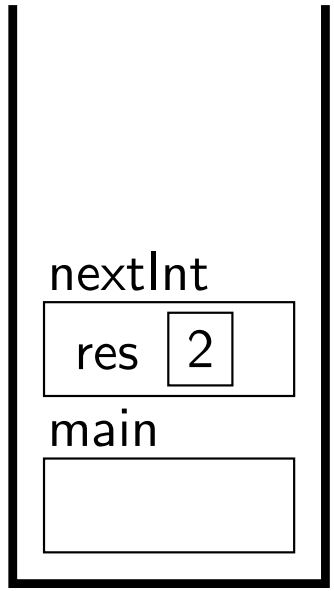
Output

- 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
```

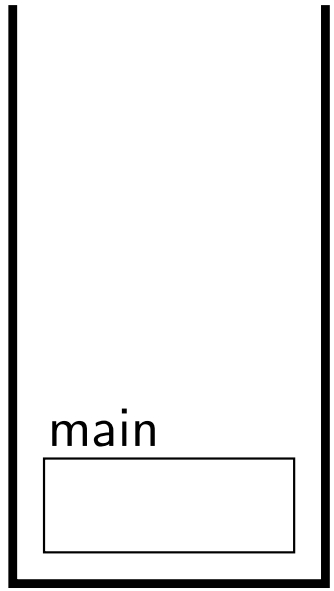
Output

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

More on Static Local Variables

```
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
```

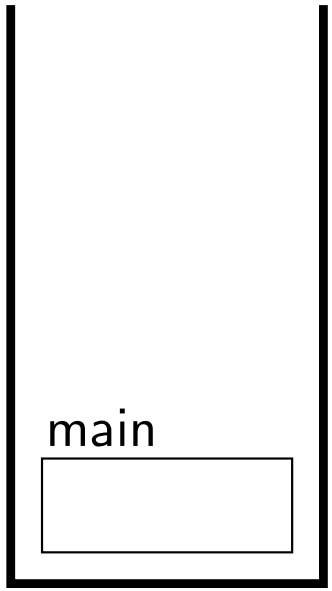
Output

- 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
next: 2
```

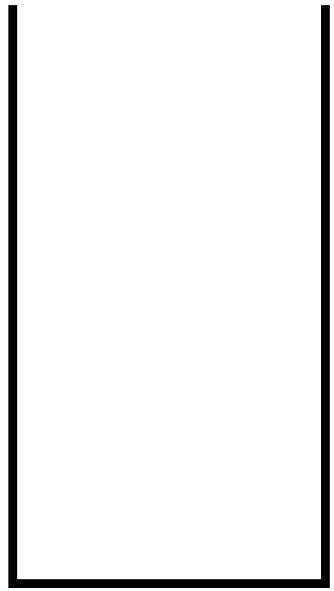
Output

- 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
next: 2
$
```

Output

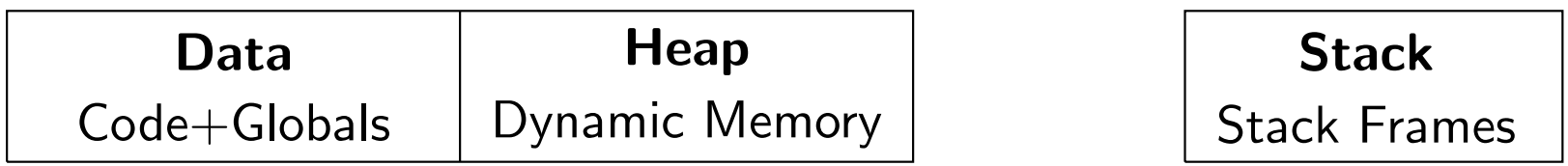
- The value remains from one call to another (initializer evaluated only once)
- This variable cannot live on the stack: would have been erased by another call
- Understanding where it lives require some more background on the system (actually, the globals are not on the stack either)

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

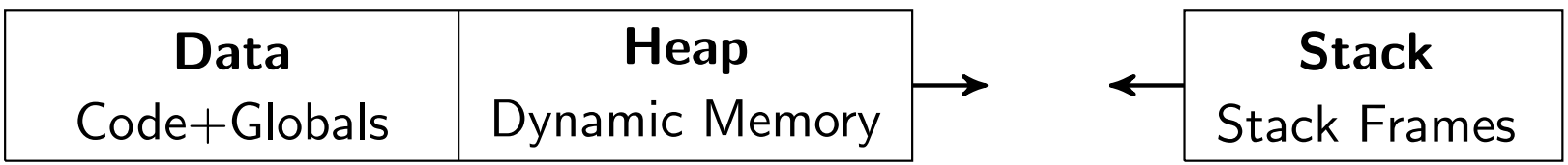


Processes Memory Layout

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

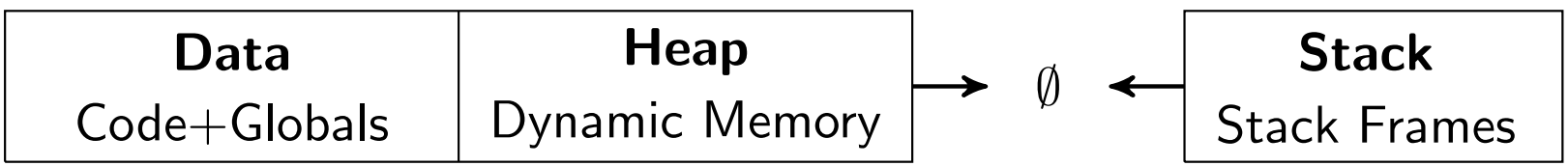
- This is a simplification, but the ideas are there



Processes Memory Layout

Primer from Next Year in System Course

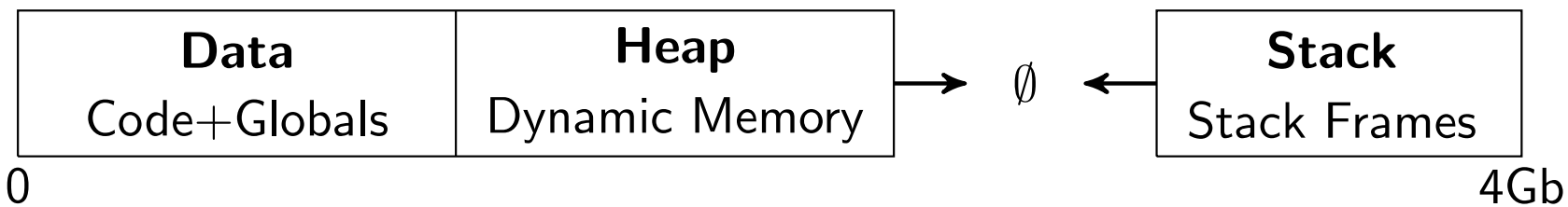
- The memory of each process is split in 3 big **segments**
- Heap is for the manually managed memory (see in half an hour)
- If more stack frames needed, the size of the stack grows toward the heap
  Conversely, the heap can grow toward the stack
- Between Heap and Stack, there is a hole
- If that hole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



Processes Memory Layout

Primer from Next Year in System Course

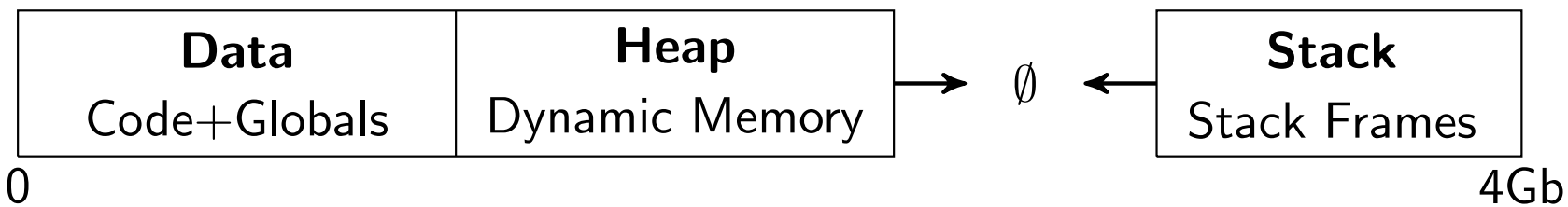
- The memory of each process is split in 3 big **segments**
- Heap is for the manually managed memory (see in half an hour)
- If more stack frames needed, the size of the stack grows toward the heap
  Conversely, the heap can grow toward the stack
- Between Heap and Stack, there is a hole in the addressing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



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

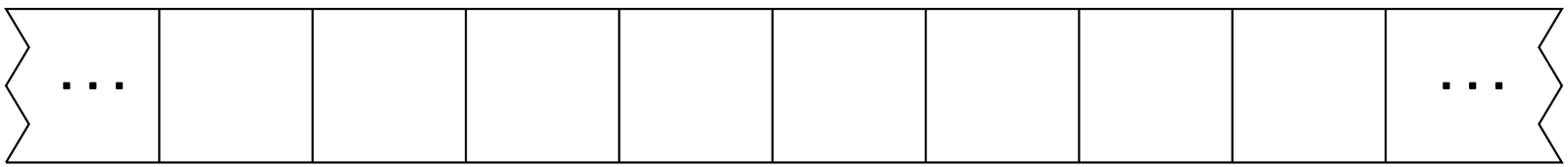
- What is the addressing space? $\leadsto$ This is another name for "memory"
- How to get a valid mental representation of the memory?

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

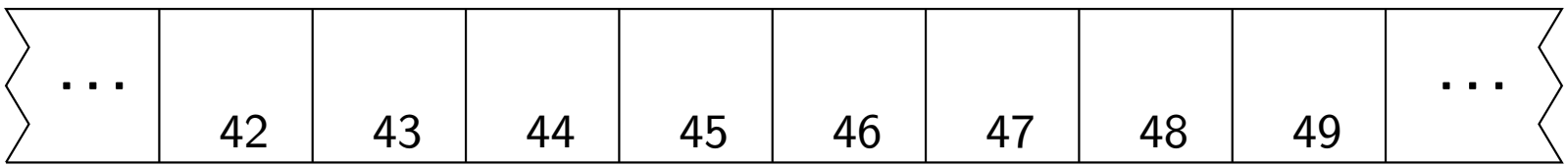


- What is an address?
- Why the stack bottom at 4Gb?
- Where is my stack if my laptop does not have 4Gb?

More on Memory

Solving the Enigma of Static Locals Storage raises New Questions

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



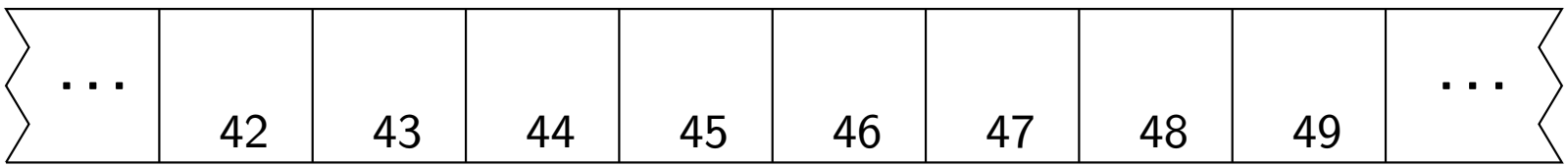
- What is an address? $\rightsquigarrow$ Memory cells are numbered.
  The **address** of a given memory cell is its number in rank
- Why the stack bottom at 4Gb?

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

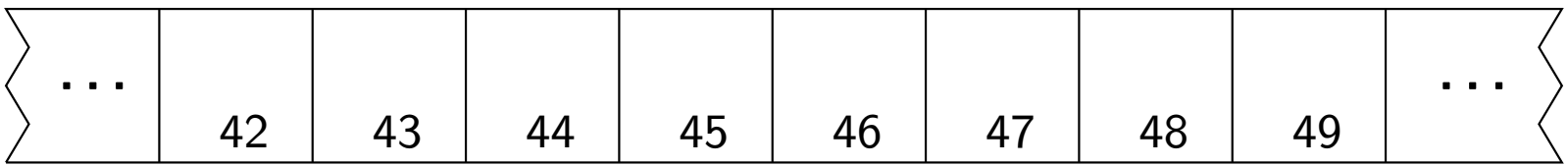


- What is an address? $\rightsquigarrow$ Memory cells are numbered.
  The **address** of a given memory cell is its number in rank
- Why the stack bottom at 4Gb? $\rightsquigarrow$ Because this is MAXINT on 32bits
  And the picture supposed that we were in 32bits for simplicity sake.
- Where is my stack 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? $\rightsquigarrow$ This is another name for "memory"
- How to get a valid mental representation of the memory?
  $\rightsquigarrow$ Think of a very large array of cells. Each cell is 1 byte (8 bits) wide.



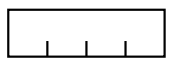
- What is an address? $\rightsquigarrow$ Memory cells are numbered.
  The **address** of a given memory cell is its number in rank
- Why the stack bottom at 4Gb? $\rightsquigarrow$ Because this is MAXINT on 32bits
  And the picture supposed that we were in 32bits for simplicity sake.
- Where is my stack 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

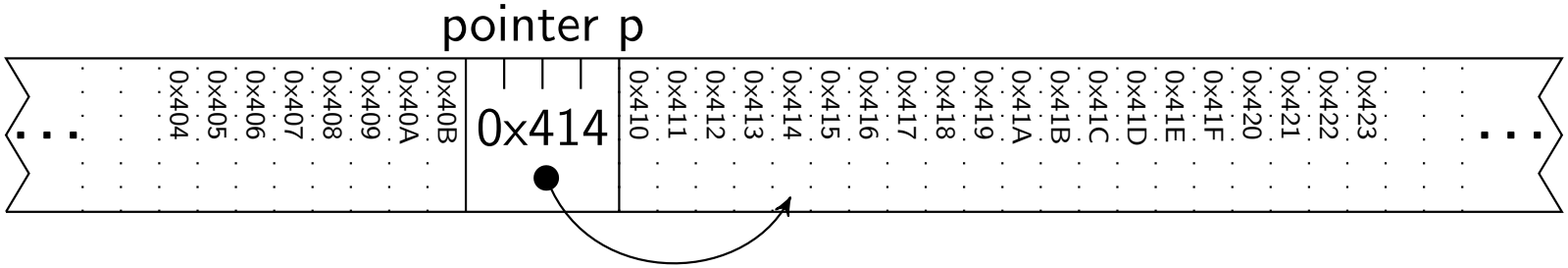
Memory Management in C

- 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?

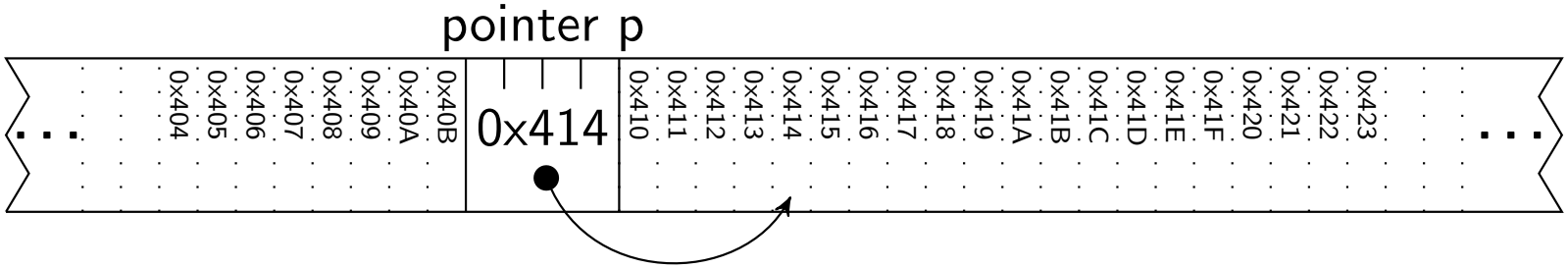
- Variable storing a memory address: Pointer value = rank of a memory cell
- On 32 bits, I need 4 bytes to store an address since biggest address$=2^{32\times 8}$ (8 bytes on 64 bits)
- Pointers are often written in hexadecimal (just a convention)
- Most of the time, numerical value is meaningless; where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



Pointers

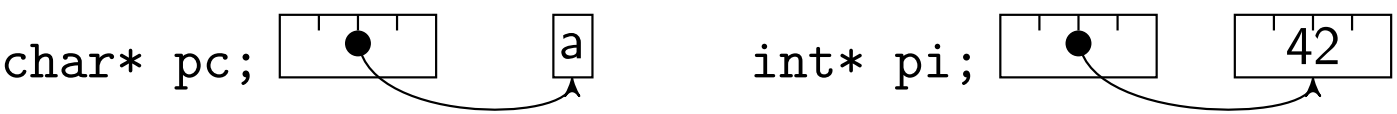
What is it?

- 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



- 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

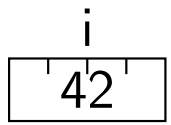
Retrieving the address of something

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;`

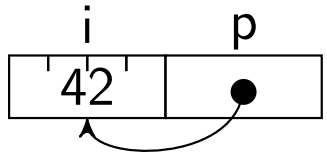


Retrieving the address of something

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;` 

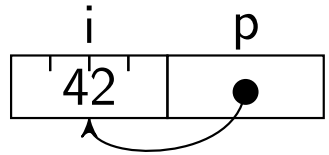
(successive variables are (often) adjacent)

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

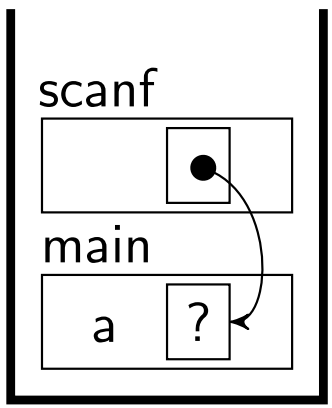
This is what the & operator does

- `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

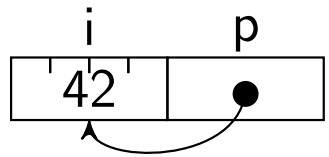
- `scanf` parameter: an address
  "%d" tells how to interpret it
- That's copied over, but that's fine
- `scanf` can modify the a variable, even if it's not in its scope (remember: C memory is a magma)
- other mystery:

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

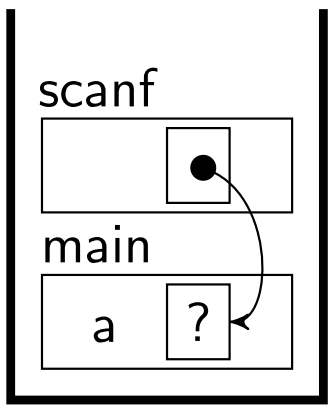
This is what the & operator does

- `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);
}
```



Stack

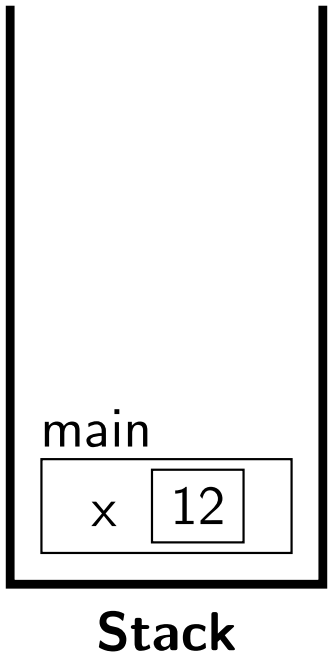
- `scanf` parameter: an address
  "%d" tells how to interpret it
- That's copied over, but that's fine
- `scanf` can modify the a variable,
  even if it's not in its scope
  (remember: C memory is a magma)
- other mystery: variable amount of params
  `man stdarg` ;)

Fixing the triple() function

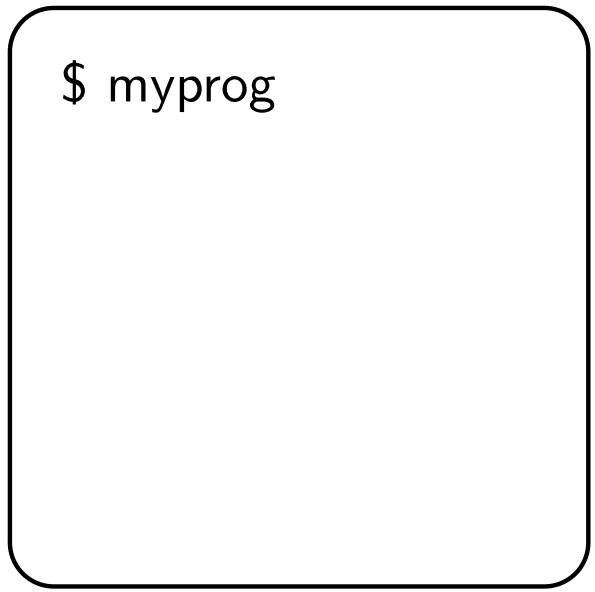
- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

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

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



Stack



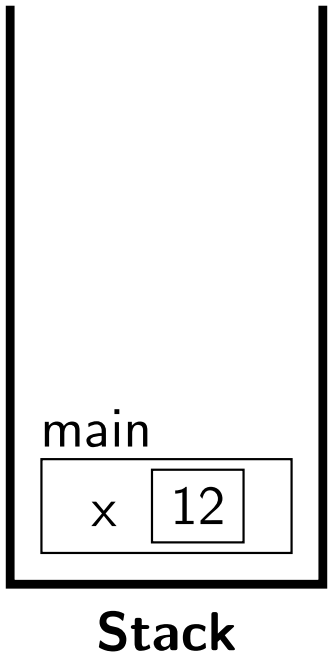
Output

Fixing the triple() function

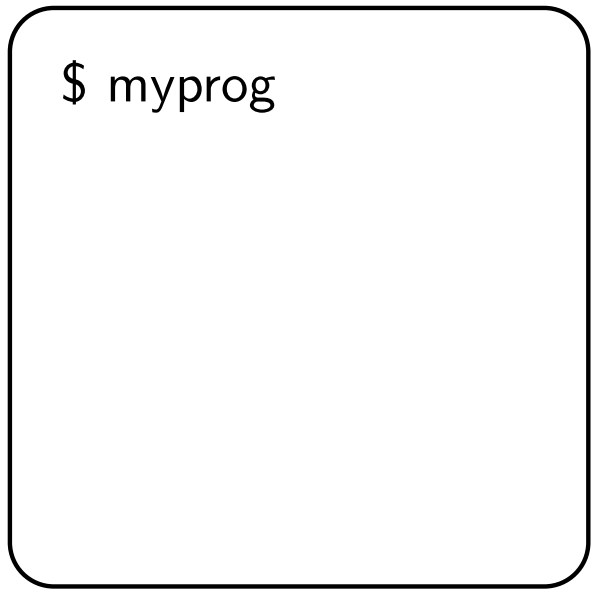
- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

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

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



Stack



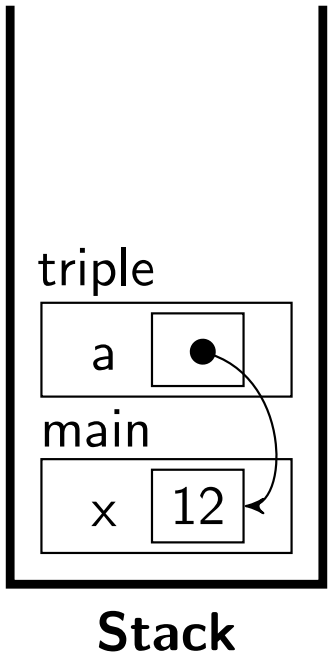
Output

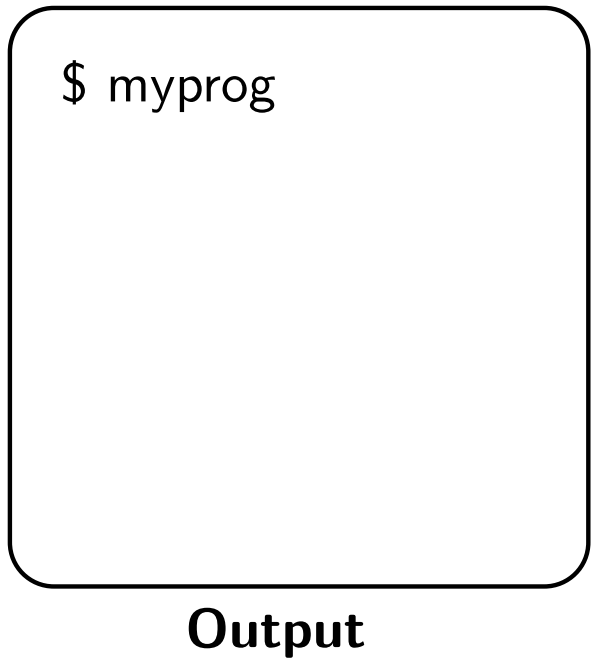
Fixing the triple() function

- 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

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

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



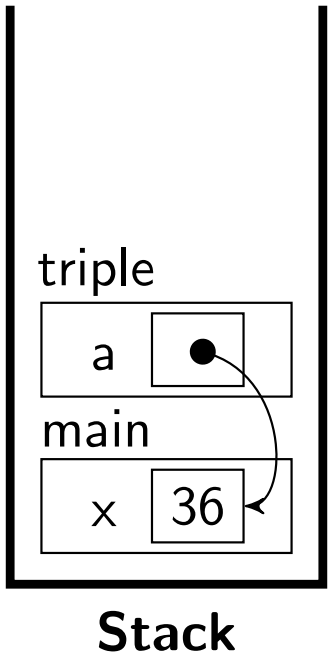


Fixing the triple() function

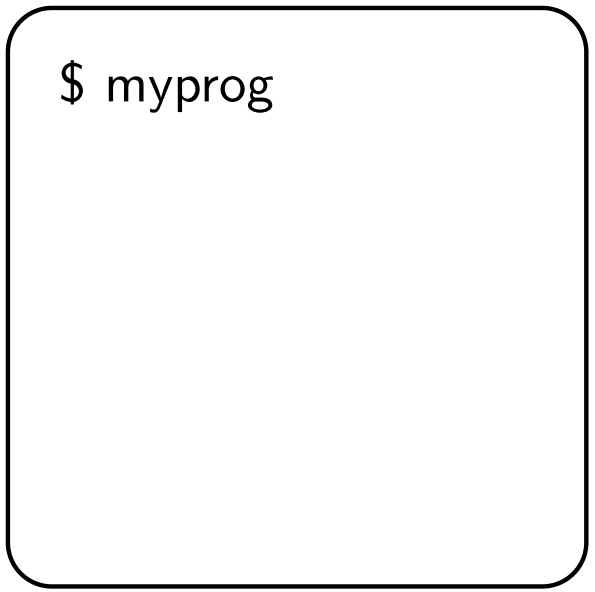
- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

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

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



Stack



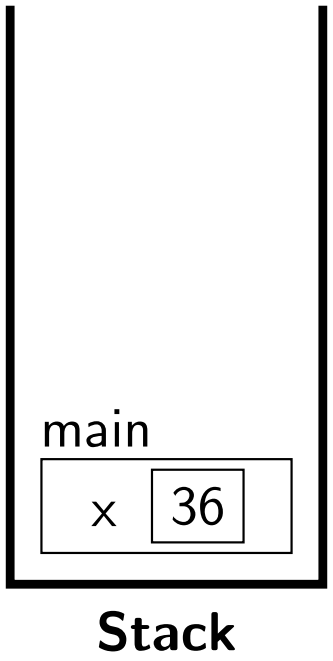
Output

Fixing the triple() function

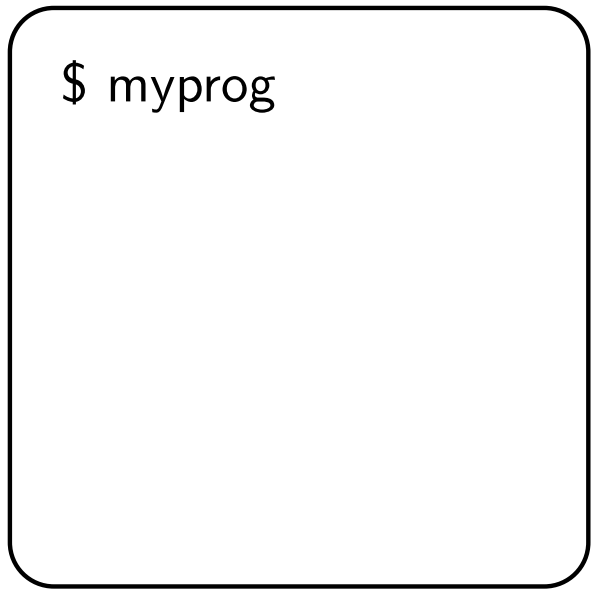
- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

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

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



Stack



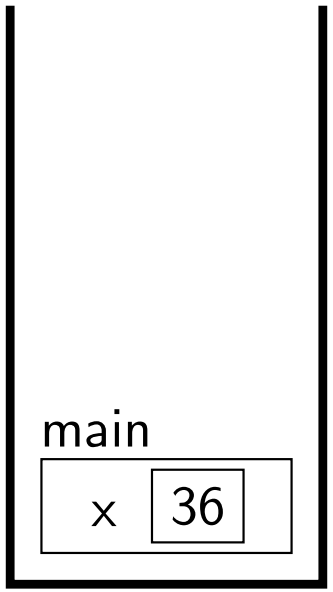
Output

Fixing the triple() function

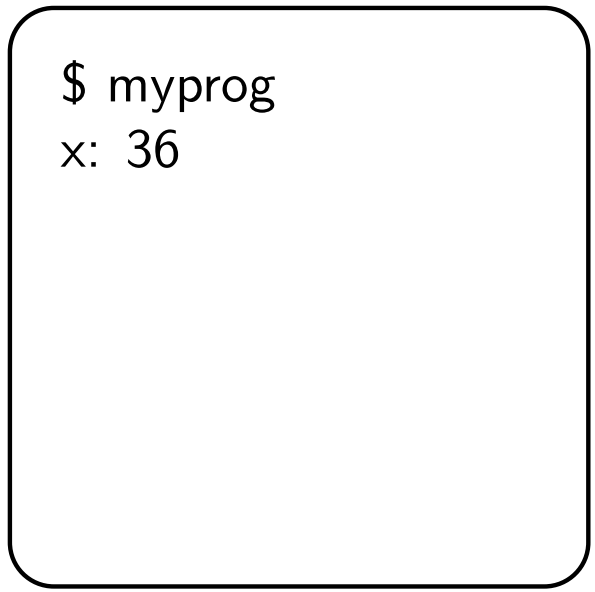
- Remember our broken triple() function, which were unable to triple its argument
- That was because parameters are passed by value (copied over)
- To fix it, we simply use a pointer parameter

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

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



Stack



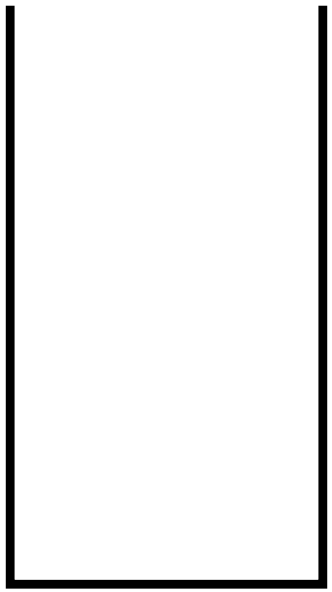
Output

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

```
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[ ])` $\approx$ `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

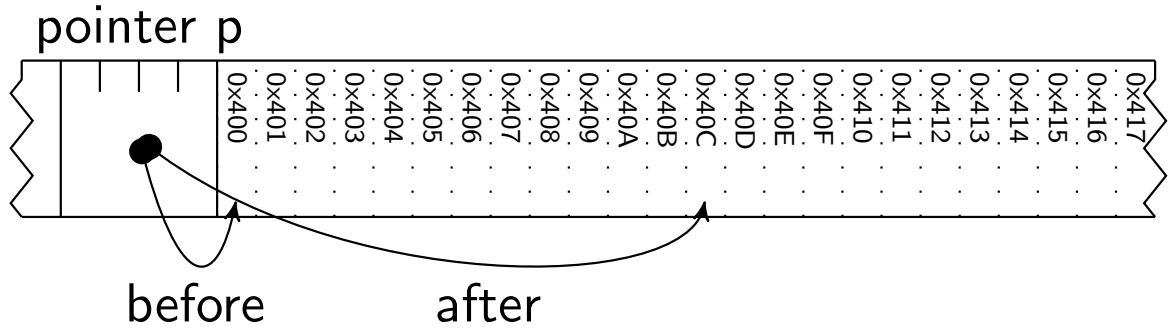
- `int *pi=...; pi[3];` This is valid; Behave as expected
  (no bound checking, as usual in C)

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

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

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 Data

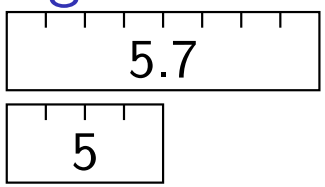
What is it?

- 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 Data

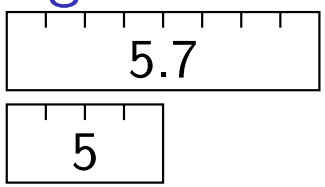
What is it?

- 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 (!)

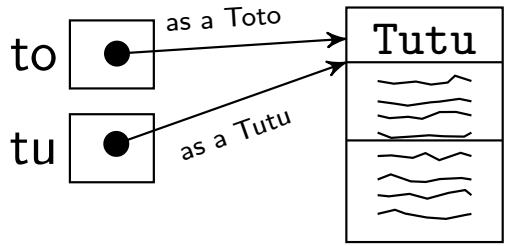
Casting Pointers: Changing the semantic

- 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

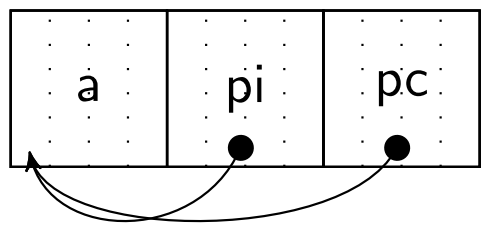
- Casts relax constraints at compilation time; Enforced at execution time
  That is what `TypeCastException` is made for
- Guessing which method gets called is sometimes excessively difficult
  Check again TD4 of POO if you forgot
- But it's hard to depreciate the Java typing system in a course on C...

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++;
```

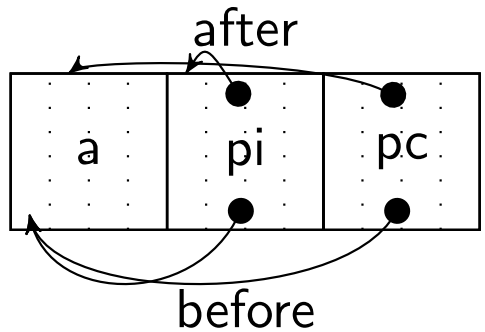


Casting Pointers in C

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

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

Dynamic Memory

Motivation

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

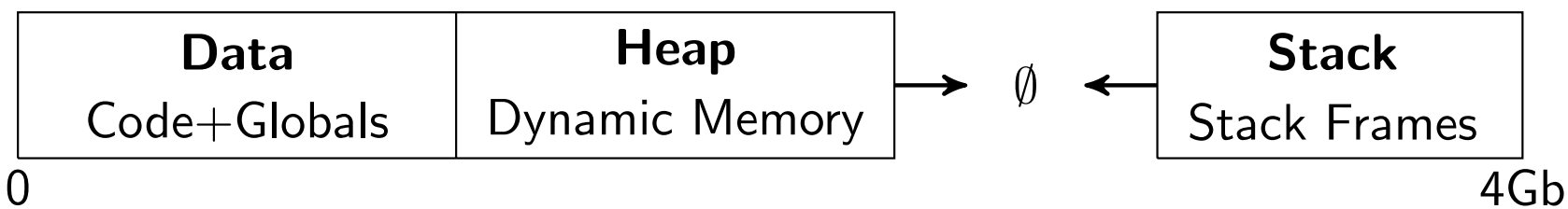
```
int n;
scanf("%d",&n);
int tab[n];
```

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



- 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

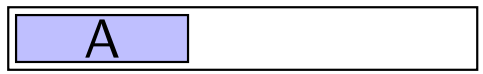
```
#include <stdlib.h>
void*malloc(int size)          Request a new memory chunk
void free(void*p)              Return a memory chunk
void*realloc(void*p,int size)  Expend a memory chunk
```

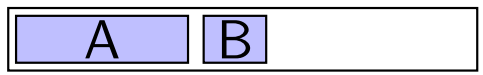
Understanding malloc and friends

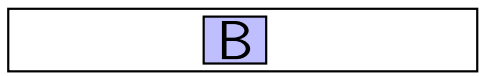
Function Semantic

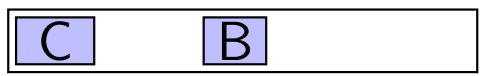
- malloc() requests a new memory chunk and return the address of beginning
  If there is not enough free memory, it returns NULL

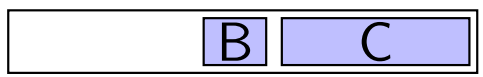
Think of a land registry for the memory

- `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

Chapter 4

Advanced C-Fu

- Modular C
  - Organizing large C projects
  - Compiling Multi-Files Projects and Makefiles
- The Many Ways of Messing Up in C
  - Syntax Pitfalls
  - Understanding gcc Error Messages
  - Messing Up With Memory
- Doing a Game in C

Organizing large C projects

- You are free in C: many ways to organize your code, nothing is enforced
- Get organized by yourself, or you'll get drown in your own code

Guidelines for Java programmers in C (Light and DIY object-orientation)

- Organize your code as several interacting classes
- Avoid inheritance by all means: ugly to mimick in C, often not helping anyway
- You can still get encapsulating and some of dynamic dispatch
- Each class becomes a module:
  - A structure, grouping all fields of your class
  - A set of functions acting on those structures (incl. constructor & destructor)
- Nothing is enforced; your code should remain clean (and your days pleasant)
  - Code readability as main objective (you are the main reader, help the reader)

Forget about the performance, genericity, reusability ... for now

> *We should forget about small efficiencies, say about 97% of the time:* **premature optimization is the root of all evil**. *Yet we should not pass up our opportunities in that critical 3%.* — Donald Knuth

The `point` module

Changing each class into a module

- A structure, grouping all fields of your class
- A set of functions acting on those structures (incl. constructor & destructor)

```
typedef struct {
  int x, y;
} point_t;
```

```
point_t *point_create(int x, int y);
void point_free(point_t *p);

void point_move(point_t *p, int dx, int dy);
void point_add(point_t *p1, point_t *p2);
```

Java	C
References to objects	Pointers to structures
Methods included in object	Functions grouped by modules Naming conventions at best
Dotted notation `p.move(3,5)`	Receiver as first parameter `point_move(p, 3,5)`
Automatic garbage collection	Manual memory handling

Full code of the `point` module

```
typedef struct {
  int x, y;
} point_t;
```

```
point_t *point_create(int x, int y) {
  point_t *res=malloc(sizeof(point_t));
  res->x = x;
  res->y = y;
  return res;
}
```

```
void point_free(point_t *p) {
  free(p);// plz still use a free function:
}         // more extensible for future
```

```
void point_move(point_t *p,
                int dx, int dy) {
  p->x += dx; // p->x shortcut of (*p).x
  p->y += dy;
}
```

```
void point_add(point_t *p1,
               point_t *p2) {
  p1->x += p2->x;
  p1->y += p2->y;
}
```

- This really feels as Java, and this is a good news:
- You can code in C and still organize your code as you've learned

- Missing: Hiding the implementation: How to have private methods and fields?
- Missing: Dynamic dispatch. Functions' pointers can simulate this.
- Missing: Inheritance. No easy way (but several ugly ones ;)

Having private methods in C modules

The File is the Compilation Unit

- Hidden methods must simply be marked `static` ⇒ visible from that file only
- This older habit explains why Java forces public classes to have their own file

How to make parts visible from outside? With **header files**!

- Regular C files named `point.h` containing structures & function prototypes
- Hide your implementation ⇒ hide the struct's content (**opaque structure**)

point.h

```
typedef struct point point_t;

point_t *point_create(int x, int y);
void point_free(point_t *p);

void point_move(point_t *p, int dx, int dy);
void point_add(point_t *p1, point_t *p2);
```

point.c

```
#include "point.h"
struct point {
  int x,y;
};

point_t *point_create(int x, int y) {
  point_t *res = malloc(sizeof(point_t));
  res->x = x;
  res->y = y;
  return res;
}
      ...
```

Dealing with the compiler's stupidity

Problem: the C compiler is really prehistoric

- Complains when symbols get redefined (even if the definitions match)
- Problem when `point.h` ∈ `square.h` ∈ `main.c` and `square.h` ∈ `main.c`
- **Multiple inclusion** of `point.h` into `main.c`, leading to compilation error

Solution: fix the code before compiling

- Remember: the preprocessor changes the code presented to the compiler
- We need to hide the subsequent inclusions of files

point.h

```
#ifndef POINT_H
#define POINT_H
typedef struct point point_t;

point_t *point_create(int x, int y);
void point_free(point_t *p);

void point_move(point_t *p, int dx, int dy);
void point_add(point_t *p1, point_t *p2);
#endif /* POINT_H */
```

- `point.c` remains unchanged
- This construct seems ugly first
- But this is the one true way
- Just works, simple and efficient
- Not sufficient on Windows.
  C on Windows is pure masochism
  (but not because of C)

Advanced OOP in C

Dynamic dispatch

- Structures can contain pointers to function (but shouldn't when possible)

```
typedef struct point *point_t; // forward decl
struct point {
  int x, y;
  void (*display)(point_t* p);
}
```

```
void my_display(point_t*p) {..}
     ...
p1->display = &my_display;
     ...
(*(p1->display)) (p1); // just a call
```

Inheritance

- Including structures is a UGLY but working approach. Don't do this for real
- Inheritance is over-sold anyway. You should never expose your inheritance tree

```
typedef struct particle {
  point_t super; // whole structure copied over
  int vx, vy;
} particle_t;
```

```
void particle_animate(particle_t *mp) {
  mp->super.x = p1->vx;
  mp->super.y = p1->vy;
}
```

- Any particle can even be casted into a point to retrieve x and y
- But NO SAFEGUARD here. So pain to debug, impossible to read, etc.
- Y U NO C++ (or Objective C) if you really need OOP in C??

Having many files in your project (one per module)

That's a good habit

- Because a 500,000 lines file is hard to navigate
- Because we can compile each of them separately
  - Remember: compilation = translation into assembly lang. + linking of ASM
  - Translation: hard and takes time; Assembling the puzzle: much faster
  - Multiple files allows to translate only the parts that changed
- Because several people can work on the same project w/o interfering

Having many files in your project (one per module)

That's a good habit

- Because a 500,000 lines file is hard to navigate
- Because we can compile each of them separately
  - Remember: compilation = translation into assembly lang. + linking of ASM
  - Translation: hard and takes time; Assembling the puzzle: much faster
  - Multiple files allows to translate only the parts that changed
- Because several people can work on the same project w/o interfering

But harder to compile right

- Compilation: `gcc -c point.c` `gcc -c square.c` `gcc -c main.c`

  It generates `point.o`, `square.o`, `main.o` containing the assembly translations
- Linking: `gcc -o project point.o square.o main.o`
- Tracking dependencies is a nightmare *e.g.* when header files are changed
- We need a specific tool for that

Having many files in your project (one per module)

That's a good habit

- Because a 500,000 lines file is hard to navigate
- Because we can compile each of them separately
  - Remember: compilation = translation into assembly lang. + linking of ASM
  - Translation: hard and takes time; Assembling the puzzle: much faster
  - Multiple files allows to translate only the parts that changed
- Because several people can work on the same project w/o interfering

But harder to compile right

- Compilation: `gcc -c point.c` `gcc -c square.c` `gcc -c main.c`

  It generates `point.o`, `square.o`, `main.o` containing the assembly translations
- Linking: `gcc -o project point.o square.o main.o`
- Tracking dependencies is a nightmare *e.g.* when header files are changed
- We need a specific tool for that. It's called **make**
- Even from eclipse, use makefiles. Obey the UNIX philosophy:

Write programs that do one thing and do it well. Write programs to work together.

Make and Makefiles

Makefile: explaining the project building process

- Create a file named `Makefile` in your project, containing a set of rules

```
<target file>: <list of dependencies>
    <command to build target from deps>
```

Simple Makefile

```
project: point.o square.o main.o
    gcc point.o square.o main.o -o project

point.o: point.c point.h
    gcc -c point.c

square.o: square.c square.h point.h
    gcc -c square.c

main.o: main.c square.h point.h
    gcc -c main.c
```

make already knows to build .o from .c

```
project: point.o square.o main.o
    gcc point.o square.o main.o -o project

point.o: point.c point.h
square.o: square.c square.h point.h
main.o: main.c square.h point.h
```

make loves funky variable names

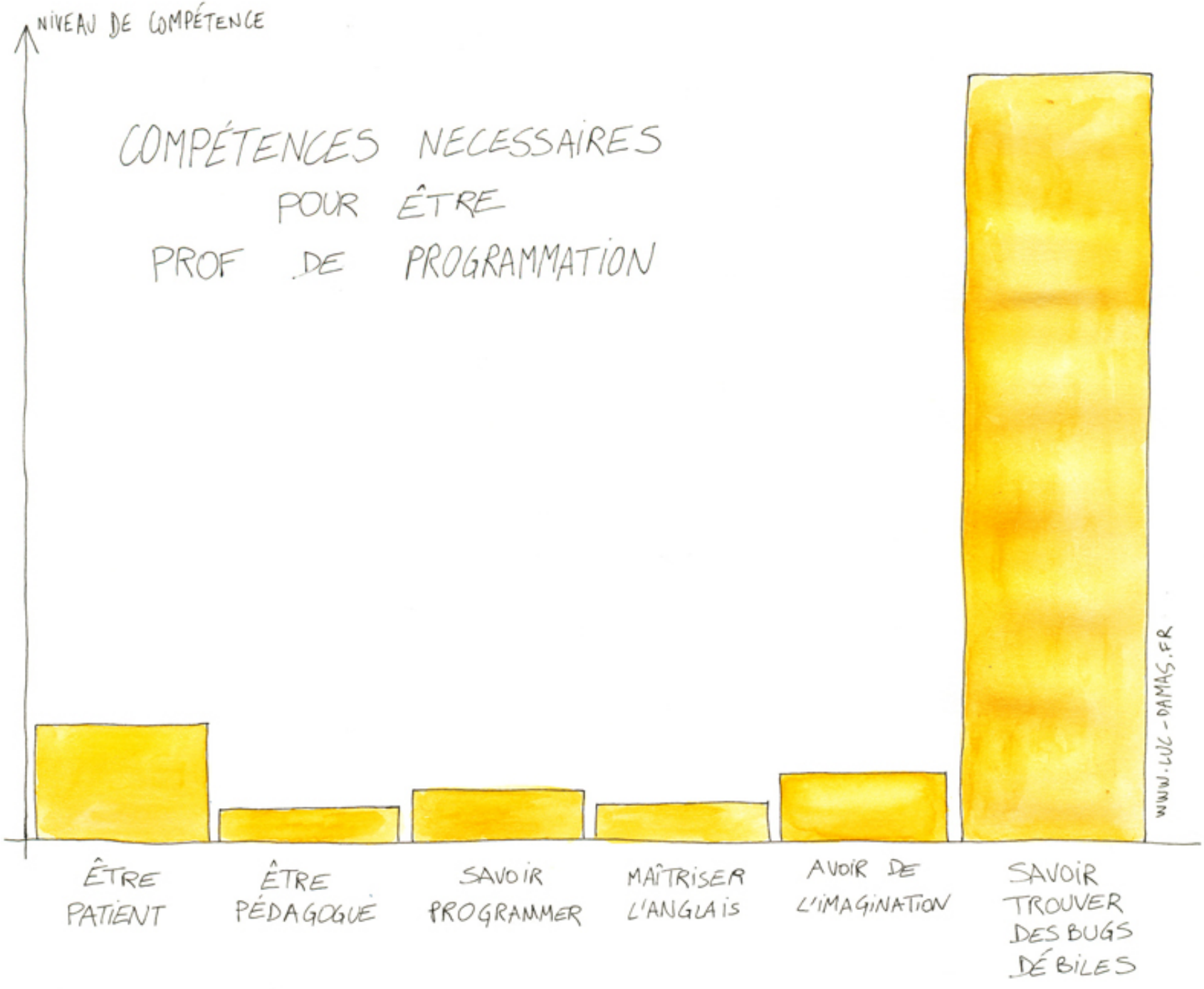
```
project: point.o square.o main.o
    gcc $^ -o $@
```

- Builds first target by default; Specify another one if you want `make clean`
- `make` is used widely, not only for C. You could use it for you Java code!

Chapter 4

Advanced C-Fu

- Modular C
  - Organizing large C projects
  - Compiling Multi-Files Projects and Makefiles
- The Many Ways of Messing Up in C
  - Syntax Pitfalls
  - Understanding gcc Error Messages
  - Messing Up With Memory
- Doing a Game in C


NIVEAU DE COMPÉTENCE
COMPÉTENCES NECESSAIRES POUR ÊTRE PROF DE PROGRAMMATION
ÊTRE PATIENT
ÊTRE PÉDAGOGUE
SAVOIR PROGRAMMER
MAÎTRISER L'ANGLAIS
AVOIR DE L'IMAGINATION
SAVOIR TROUVER DES BUGS DÉBILES
WWW.LUC-DAMAS.FR

Classical errors with the `for` loop

```
for (i=0; i < 10; i+1)
   printf("i=%d\n",i);
```

Classical errors with the `for` loop

```
for (i=0; i < 10; i+1)
   printf("i=%d\n",i);
```

```
for (i=0; i = 10; i++)
   printf("i=%d\n",i);
```

- Y U NO increment your counter??
  (`i+1` has no side effect)

Classical errors with the `for` loop

```
for (i=0; i < 10; i+1)
   printf("i=%d\n",i);
```

- Y U NO increment your counter??
  (`i+1` has no side effect)

```
for (i=0; i = 10; i++)
   printf("i=%d\n",i);
```

- Y U NO test your counter??
  (`i=10` sets a new value)

```
for (i=0; i < 10; i++);
   printf("i=%d\n",i);
```

Classical errors with the `for` loop

```
for (i=0; i < 10; i+1)
   printf("i=%d\n",i);
```

- Y U NO increment your counter??
  (`i+1` has no side effect)

```
for (i=0; i = 10; i++)
   printf("i=%d\n",i);
```

- Y U NO test your counter??
  (`i=10` sets a new value)

```
for (i=0; i < 10; i++);
   printf("i=%d\n",i);
```

- Y U NO enter your loop??
  (the `;` after the `for` closes the loop)

```
for (i=0, j=0; i < 10; i++)
   printf("i=%d\n",i);
```

Classical errors with the `for` loop

```
for (i=0; i < 10; i+1)
   printf("i=%d\n",i);
```

- Y U NO increment your counter??
  (`i+1` has no side effect)

```
for (i=0; i = 10; i++)
   printf("i=%d\n",i);
```

- Y U NO test your counter??
  (`i=10` sets a new value)

```
for (i=0; i < 10; i++);
   printf("i=%d\n",i);
```

- Y U NO enter your loop??
  (the `;` after the `for` closes the loop)

```
for (i=0, j=0; i < 10; i++)
   printf("i=%d\n",i);
```

- Y U NO see when it's correct??
  `,` separate expressions, `;` instructions

Beware of the vicious `switch` syntax!

```
int x = 2;
switch(x) {
  case 2:
    printf("Two\n");
  case 3:
    printf("Three\n");
}
```

Beware of the vicious `switch` syntax!

```
int x = 2;
switch(x) {
  case 2:
    printf("Two\n");
  case 3:
    printf("Three\n");
}
```

- Prints both:

```
Two
Three
```

- Problem: missing `break` keywords
- Because in assembly, that's a jump table

- So that's a (sad) inheritance of assembly language
- And that's very sad that this propagated to Java...
- This also explains why `case` values must be constant

Understanding `gcc` error messages

`gcc` **is** user friendly. It's just picky about its friends

But you **have to** pass `-Wall -Wextra` as parameter

Suggest parentheses around assignment used as truth value...

```
if (a=b)
```

Understanding `gcc` error messages

`gcc` **is** user friendly. It's just picky about its friends

But you **have to** pass `-Wall -Wextra` as parameter

Suggest parentheses around assignment used as truth value...

```
if (a=b)
```

- ...if you **really** mean to erase a, then write `if ((a=b))`
- Else, you probably meant `if (a==b)`
- The compiler gives a meaning even to the weird `if ((a=b)!=0)`

Return of functions

- warning: 'return' with a value, in function returning void
  Yeah that's only a warning — keep calm and add `-Werror`

Return of functions

- warning: 'return' with a value, in function returning void
  Yeah that's only a warning — keep calm and add `-Werror`

- Control reaches end of non void function: self explanatory (?)

Don't do that

```
int my_function() {
   if (x == 2)
      return 1;
}
```

Return of functions

- warning: 'return' with a value, in function returning void
  Yeah that's only a warning — keep calm and add `-Werror`
- Control reaches end of non void function: self explanatory (?)

Don't do that

```
int my_function() {
   if (x == 2)
      return 1;
}
```

This is correct

```
int my_function() {
   if (x == 2)
      return 1;
   return 0;
}
```

This is correct too

```
void my_function() {
   if (x == 2)
      printf("blah");
}
```

Declarations

Implicit declaration of function func

- The function was used before being declared
- Just warning; the creepy compiler assumes "no parameter, returning integer"
- If you declare the function **after** its use, the message reads:

  warning: conflicting types for 'func'
  - and you are informed of where the "declaring usage" occured

Too few/many arguments to function

- Good programmers have a rare ability: they *try* to read error messages!

Passing arg n of func from incompatible pointer type

- Seems innocuous, but often denotes (upcoming) subtle issue

Passing arg n of func makes pointer from integer without a cast

- The numerical value of pointers should not be messed with as in

```
int value=42;
int length=strlen(value);
```

Messing Up With Memory

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
  char *truc = "constant string";
  *truc = 'x';            // segfault
  strcpy(truc, "toto"); // segfault
  free(truc);             // segfault
  return 0;
}
```

```c
int *make_buff(int a) {
  int buff[SIZE], cpt;
  for (cpt=0; cpt<SIZE; cpt++)
    buff[cpt] = a;
  return buff; // pointer to invalid memory
}
```

```c
  char *buffer;
  scanf("%s", buffer); // invalid write
```

```c
  int i = 42;
  printf("%s",i); // invalid read
```

```c
  char buffer[256];       // constants
  for (i=0; i<1024; i++) //   shouldn't
     buffer[i] = ' ';    //     change
```

But this aint fun: messing up with pointers is too easy

we'll see in practical how to hunt down these issues with valgrind

Chapter 4

Advanced C-Fu

- Modular C
  - Organizing large C projects
  - Compiling Multi-Files Projects and Makefiles
- The Many Ways of Messing Up in C
  - Syntax Pitfalls
  - Understanding gcc Error Messages
  - Messing Up With Memory
- Doing a Game in C

Doing a game in C

Mandatory elements

- A gameplay: an idea about what your game will be
- A game engine: a program that can interact with the player
- Some GFX: graphics, sounds, musics, etc.
- These may not suffice for a great game, but at least they are mandatory

Where to find the inspiration for your gameplay

- Kongregate, play.google.com or whatever.
- Many links on `http://www.loria.fr/~quinson/Hacking/Curiosa/`

Doing a game engine

- It's actually easy (with SDL2, SFML or allegro)! **That's your assignment**
- See `http://www.loria.fr/~vthomas/enseignement/2013_JV_ESIAL/`

Finding some GFX

- That's **not** your assignment. It's sufficient to find something online
- But please don't *steal* your GFX. Free resources exist.

Logistics of the project

- Groups: 2 or 3 peoples (not 1, not 4). You can mix classes
- When: first week empty of lectures at Telecom Nancy ($\approx$ end May)
- What: discuss per email with me so that we agree on an assignment
- How long: don't assume you can do something in 2 weeks only
- Other constraints:
  - Must be in C (or C++ if good reasons), under Linux (C is **NOT** portable)
  - Project must be hosted in SVN or GIT somewhere.
  - Must be rather original & involve some coding (no mastermind/memory please)
  - Should induce a graphical interface, may contain an AI
  - That's very open: do what you would like to (one-time offer in your scholarship)
- What gets evaluated: report (5 pages max) + source code + oral defense
  - A short public presentation: 10 lines description + screenshot + licensing info
  - List of issues encountered and your solutions (no code!)
  - Approximate amount of time spent per student and per task
  - Exhaustive list of source of informations you've used
- You **must** get my **written** approval before starting.
  - Try talking to me after my teachings if I don't answer my emails in time
  - Don't wait the week before the defenses or I'll know ;)