

C Preprocessor (CPP)

- Processes files before the compiler
- Lines that begin with #
- Preprocessor doesn't "know" C
- To see CPP output: `gcc -E my_file.c`
- Useful for
 - File inclusion
 - Constants
 - Macros
 - Conditional compilation

File Inclusion

- CPP replaces `#include` with the entire contents of the included file
- System headers
 - `#include <stdio.h>`
 - On UNIX, looks in `/usr/include`
 - You should not use `<>` to include user-defined header files
- User-defined headers
 - `#include "my_pow.h"`
 - Searches current (source) directory first
 - Include files can be nested
- `-I` switch used to specify additional include directories

Constants

- Simple textual replacement

```
#define name replacement text
```

- Examples:

```
#define PI 3.14
```

```
#define MAX 1000
```

```
#define TRUE 1
```

```
#define SECS_PER_DAY (60 * 60 * 24)
```

```
#define READ_ACCESS 0x01
```

```
#define WRITE_ACCESS 0x02
```

```
#define RW_ACCESS (READ_ACCESS | WRITE_ACCESS)
```

```
#undef PI /* un-defines a constant/macro */
```

Macros

```
#define max(A, B)      ((A) > (B) ? (A) : (B))
```

```
x = max(p+q, r+s);
```

becomes

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

- Notice, expressions are evaluated twice!
- `stdio.h` defines the following macros:

```
#define getchar()     getc(stdin)
```

```
#define putchar(c)    putc((c), stdout)
```

- Don't put a space between macro name and '('
- Don't end with semicolon

Stringification

`#parameter_name quotes parameter_name`

```
#define PRINT_STR_VAR(S) printf(#S " = %s\n", S)
char szDrink[] = "Okocim";
PRINT_STR_VAR(szDrink);
```

- Becomes

```
printf("szDrink" " = %s\n", szDrink);
```

- which, after string concatenation, becomes

```
printf("szDrink = %s\n", szDrink);
```

- finally, the output is

```
szDrink = Okocim
```

Stringify Constants

```
typedef enum { red, green, blue, white, black } EColor;

#define COLOR_CASE(x) case x: return "The color is " #x

const char* Color2Name(EColor clr) {
    switch (clr) {
        COLOR_CASE(red);
        COLOR_CASE(green);
        COLOR_CASE(blue);
        COLOR_CASE(white);
        COLOR_CASE(black);
        default: return "INVALID COLOR!";
    }
}
```

Stringification

- If you want to stringify the result of expansion of a macro argument, you have to use two levels of macros.

```
#define XSTR(S) STR(S)
```

```
#define STR(S) #S
```

```
#define FOO 4
```

```
STR (FOO)
```

```
==> "FOO"
```

```
XSTR (FOO)
```

```
==> XSTR (4)
```

```
==> STR (4)
```

```
==> "4"
```

```
#define FIELDSIZE 4
```

```
char* str="long string";
```

```
printf("%" XSTR(FIELDSIZE) "s",str);
```

Concatenation

- To concatenate two macro arguments, use the `##` operator:

```
#define DOUBLE(A,B) A##B
printf("%f\n",DOUBLE(3,e6));
==> printf("%f\n",3e6);
```

```
#define DOUBLE(A,B) AB
printf("%f\n",DOUBLE(3,e6));
==> printf("%f\n",AB); /* invalid */
```

```
#define DOUBLE(A,B) A B
printf("%f\n",DOUBLE(3,e6));
==> printf("%f\n",3 e6); /* invalid */
```


Pre-defined Names

- `__LINE__` Current source line number
- `__FILE__` Name of file being compiled
- Useful for error handling, debugging and logging

```
#define DEBUG_STR(S) \  
    printf(#S " = %s [File: %s, Line: %d]\n", \  
    S, __FILE__, __LINE__);
```

```
DEBUG_STR(szDrink)
```

- the output is

```
szDrink = Okocim [File: debug_str.c, Line: 16]
```

Conditional Compilation

- `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`
- Useful for separating debug-only code from release code
- `assert()`, `debug_alloc`, etc.

```
#ifdef MY_DEBUG
#define DEBUG_STR(S) \
    printf(#S " = %s [File: %s, Line: %d]\n", \
        S, __FILE__, __LINE__);
```

```
#else
```

```
#define DEBUG_STR(S)
    /* do nothing in release builds */
#endif /* MY_DEBUG */
```

- Use the `-Dmacro=value` compiler switch:

```
$ gcc -DMY_DEBUG=1 debug_str.c -o debug_str
  && ./debug_str
```

```
szDrink = Okocim [File: debug_str.c, Line: 16]
```

Macros for Portability

- Portability: Compiler-, Processor-, OS- or API- specific code

```
#ifdef _WIN32
OutputDebugString(szOutMessage);
#else /* _WIN32 */
_CrtOutputDebugString(szOutMessage);
#endif /* _WIN32 */
```

Almost a Function

- To define a macro which can be used as a function in all contexts use the following form

```
#define fun(x) do { printf("#x " "=%d\n", x); x++; } while (0)
```

- It will work in the following example requiring the semicolon after the function/macro call

```
if (condition)
    fun(t);
else
    fun(z);
```

- A simpler construct below will not work

```
#define fun(x) { printf("#x " "=%d\n", x); x++; }
```

Problems in Large Programs

- Small programs → single file
- “Not so small” programs:
 - Many lines of code
 - Multiple components
 - More than one programmer
- Problems:
- Long files are harder to manage (for both programmers and machines)
 - Every change requires long compilation
 - Many programmers cannot modify the same file simultaneously
 - Division to components is desired

Multiple Modules

- Divide project to multiple files
 - Good division to components
 - Minimum compilation when something is changed
- Easy maintenance of project structure, dependencies and creation
- Programming language constructs used to refer to external files (**extern**)
- Compilation can link several files together to create single application from several files
- Can compile several source files, link to pre-compiled object files or combination of these options

Multiple Modules

```
/* greetings.h */  
void greetings(void);
```

```
/* main.c */  
#include <stdio.h>  
#include "greetings.h"  
  
int main()  
{  
    greetings();  
    return 0;  
}
```

```
/* greetings.c */  
#include <stdio.h>  
#include "greetings.h"  
  
void greetings(void)  
{  
    printf ("Hello user !\n");  
}
```

```
gcc -g -Wall -pedantic -c main.c -o main.o  
gcc -g -Wall -pedantic -c greetings.c -o greetings.o  
gcc -g main.o greetings.o -o hello
```

compilation

linking

Multiple Modules - Change in greetings.c

```
/* greetings.h */  
void greetings(void);
```

```
/* main.c */  
#include <stdio.h>  
#include "greetings.h"  
  
int main()  
{  
    greetings();  
    return 0;  
}
```

```
/* greetings.c */  
#include <stdio.h>  
#include <stdlib.h>  
#include "greetings.h"  
  
void greetings(void)  
{  
    printf ("Hello %s!\n",  
           getenv ("LOGNAME"));  
}
```

```
gcc -g -Wall -pedantic -c greetings.c -o greetings.o  
gcc -g main.o greetings.o -o hello
```


Multiple Modules - More Changes

```
/* greetings.h */  
int greetings(void);
```

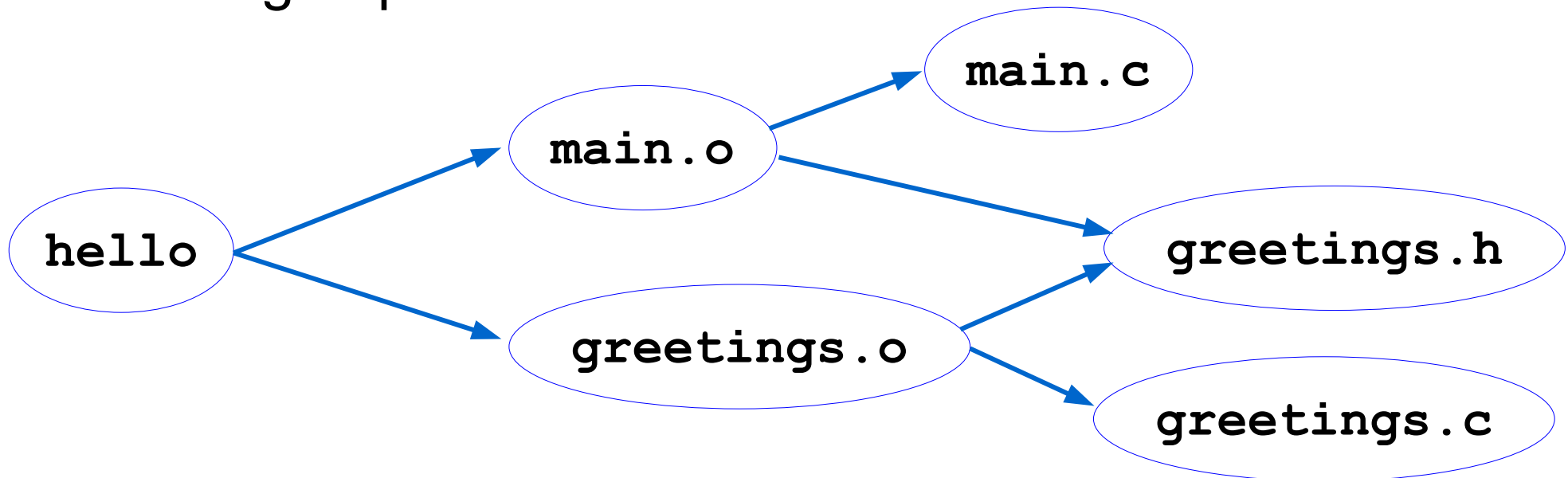
```
/* main.c */  
#include <stdio.h>  
#include "greetings.h"  
  
int main()  
{  
    greetings();  
    return 0;  
}
```

```
/* greetings.c */  
#include <stdio.h>  
#include <stdlib.h>  
#include "greetings.h"  
  
int greetings(void)  
{  
    return printf ("Hello %s!\n",  
        getenv ("LOGNAME"));  
}
```

```
gcc -g -Wall -pedantic -c main.c -o main.o  
gcc -g -Wall -pedantic -c greetings.c -o greetings.o  
gcc -g main.o greetings.o -o hello
```

Module Dependencies

- Module A has a compile-time dependency on Module B if **A.c** requires **B.h** to compile
- Program A has a link-time dependency on Module B if A requires **B.o** to link
- Avoid cyclic dependencies ($A \rightarrow B$ and $B \rightarrow A$)
- Managing software complexity in a large part requires minimizing dependencies



Makefiles and make

- Its easy to change a header file but forget to re-compile all the source files that depend on it
- Makefiles help you manage dependencies
- **make** is a utility typically used for building software packages that are comprised of many source files
- determines automatically which pieces need to be rebuilt
- uses an input file (usually called **makefile** or **Makefile**) which specifies rules and dependencies for building each piece
- you can use any name for the makefile and specify it on the command line:
 - **bash# make**
 - **bash# make -f myfile.mk**
- first way (above) uses default (**makefile** or **Makefile**) as input file
- second way uses **myfile.mk** as input file

Simple makefile

```
hello: main.o greetings.o ← dependency line
    gcc -g main.o greetings.o -o hello ← action line
main.o: main.c greetings.h
    gcc -g -Wall -pedantic -c main.c -o main.o } Dependency line
                                                + action line = rule
← Dependency line must start in column 1
greetings.o: greetings.c greetings.h
    gcc -g -Wall -pedantic -c greetings.c -o greetings.o
Action line must begin with a tab character
```

```
$ make
```

```
gcc -g -Wall -pedantic -c main.c -o main.o
```

```
gcc -g -Wall -pedantic -c greetings.c -o greetings.o
```

```
gcc -g main.o greetings.o -o hello
```

```
$ make
```

```
make: `hello' is up to date.
```

#include Guards

- When included header files include other files, some header files may be included multiple times
 - Sometimes harmless
 - e.g. multiple `extern` variable declaration
 - Sometimes harmful
 - e.g. you must not `typedef` the same type twice
 - always increases processing time
- To prevent such situations [#include guards](#) are used

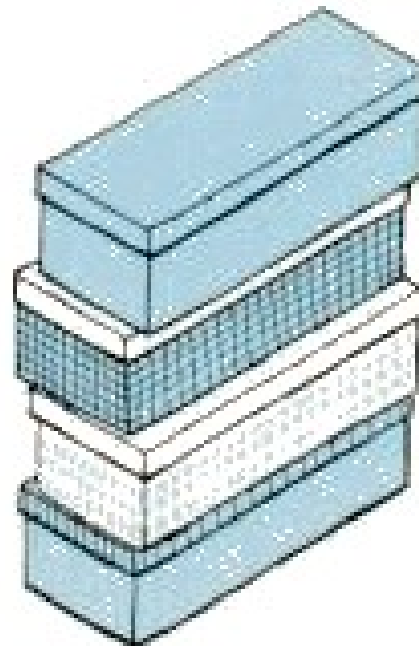
```
/* header.h */
#ifndef _HEADER_H_
#define _HEADER_H_
/* header body */
typedef some_type some_name;
#endif /* _HEADER_H_ */
```

Stacks in Our Life

A stack of cafeteria trays



A stack of pennies

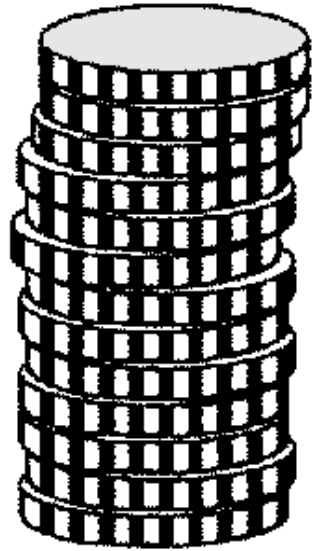


A stack of shoe boxes

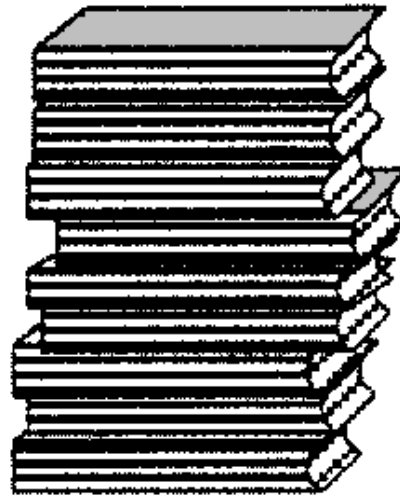
A stack of neatly folded shirts



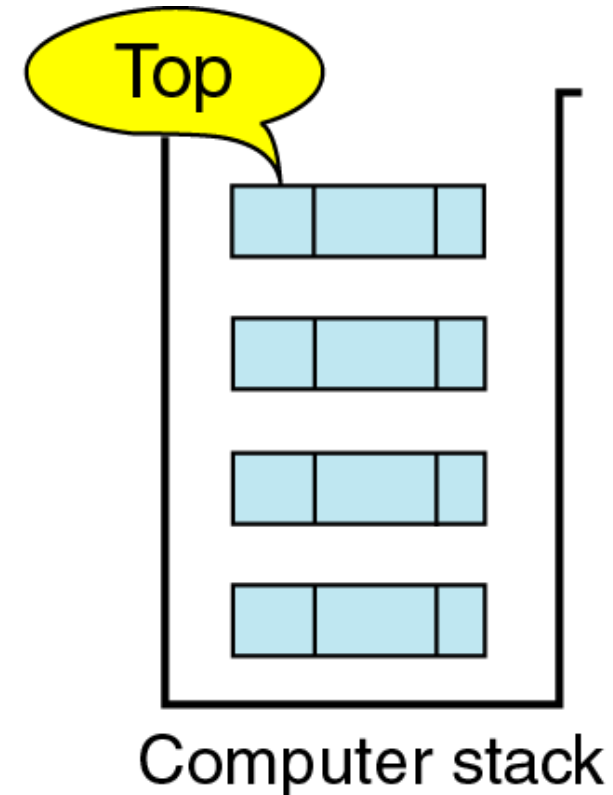
More Stacks



Stack of coins



Stack of books

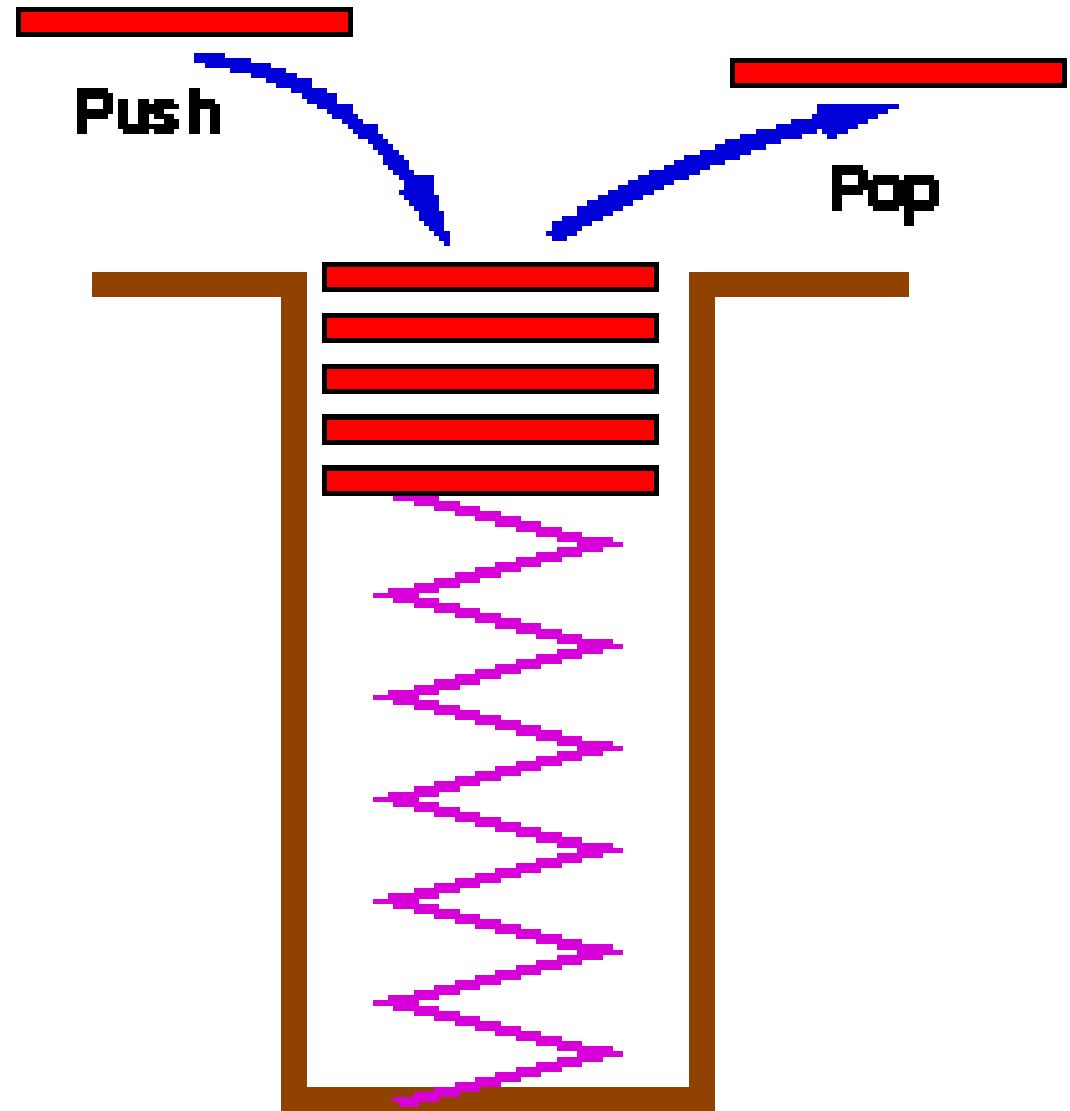


Computer stack

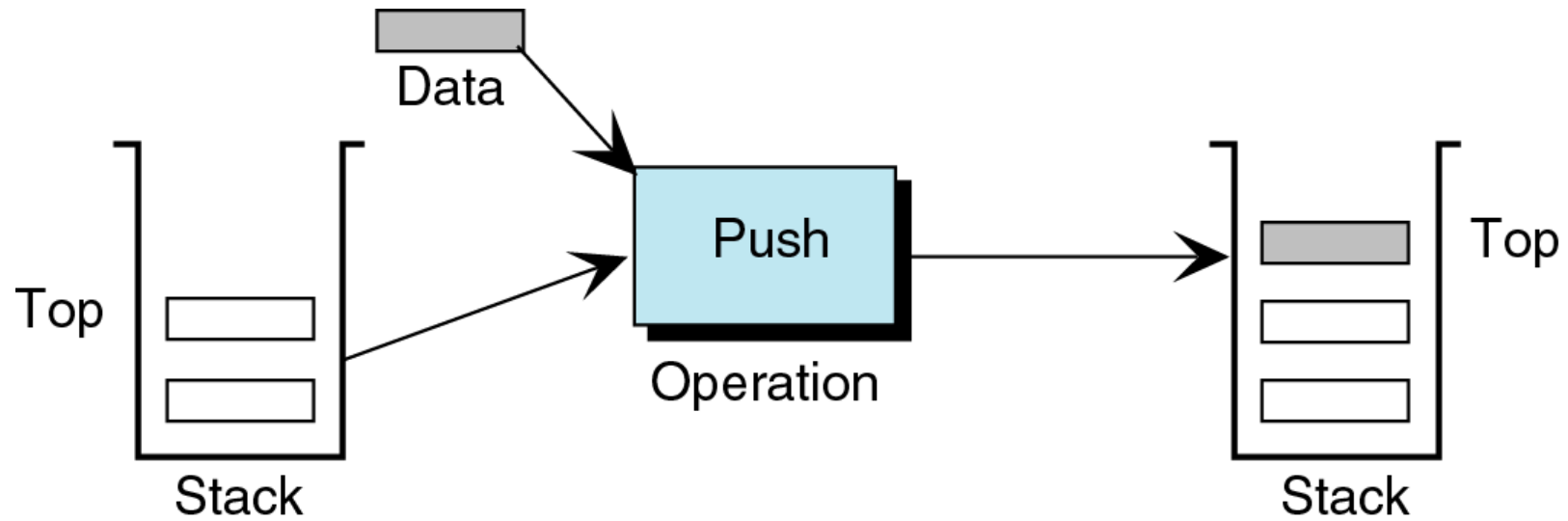
- A stack is a LIFO structure: Last In First Out

Basic Operations with Stack

- Push
 - Add an item
 - Overflow
- Pop
 - Remove an item
 - Underflow
- Stack Top
 - What's on the Top
 - Could be empty

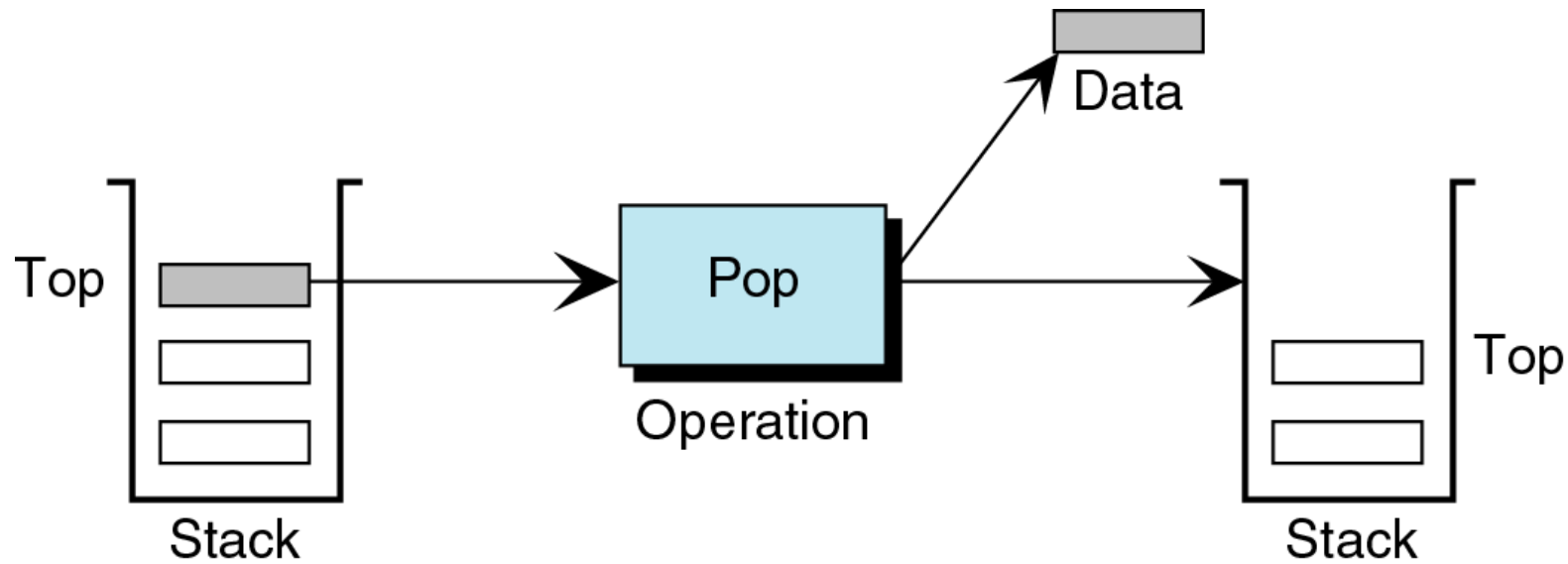


Push



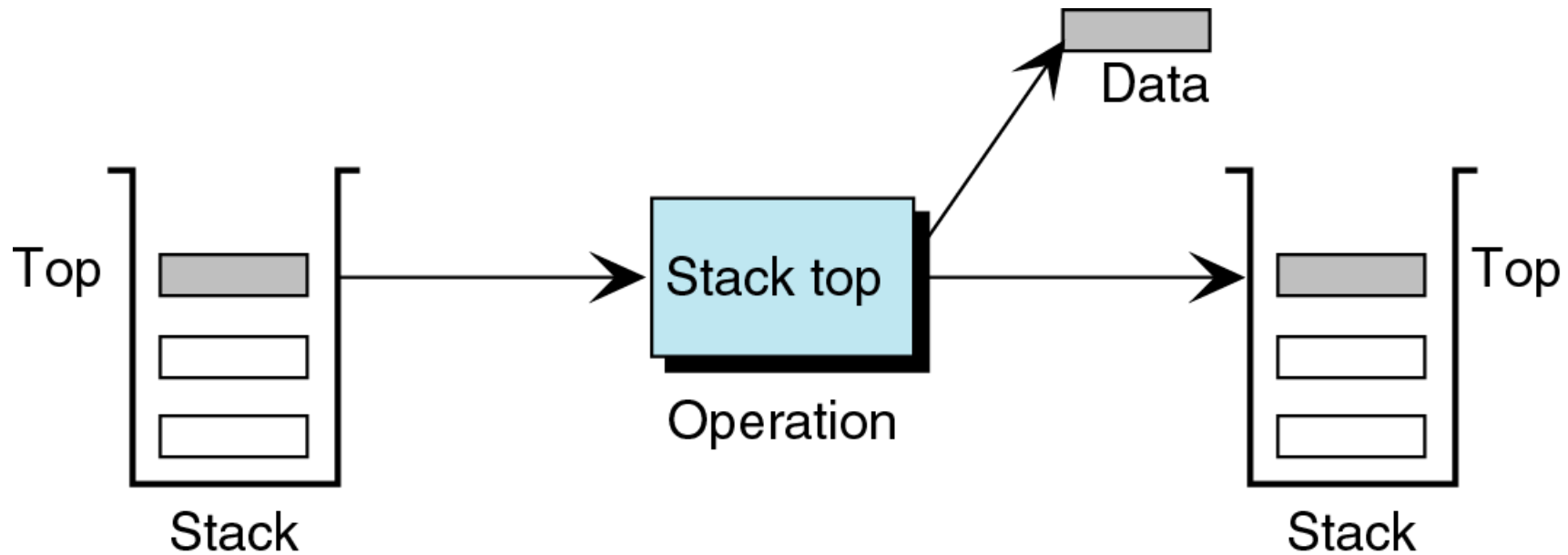
- Adds new data element to the top of the stack

Pop



- Removes a data element from the top of the stack

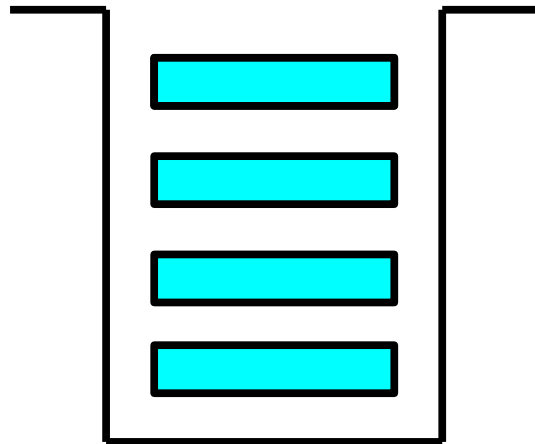
Stack Top



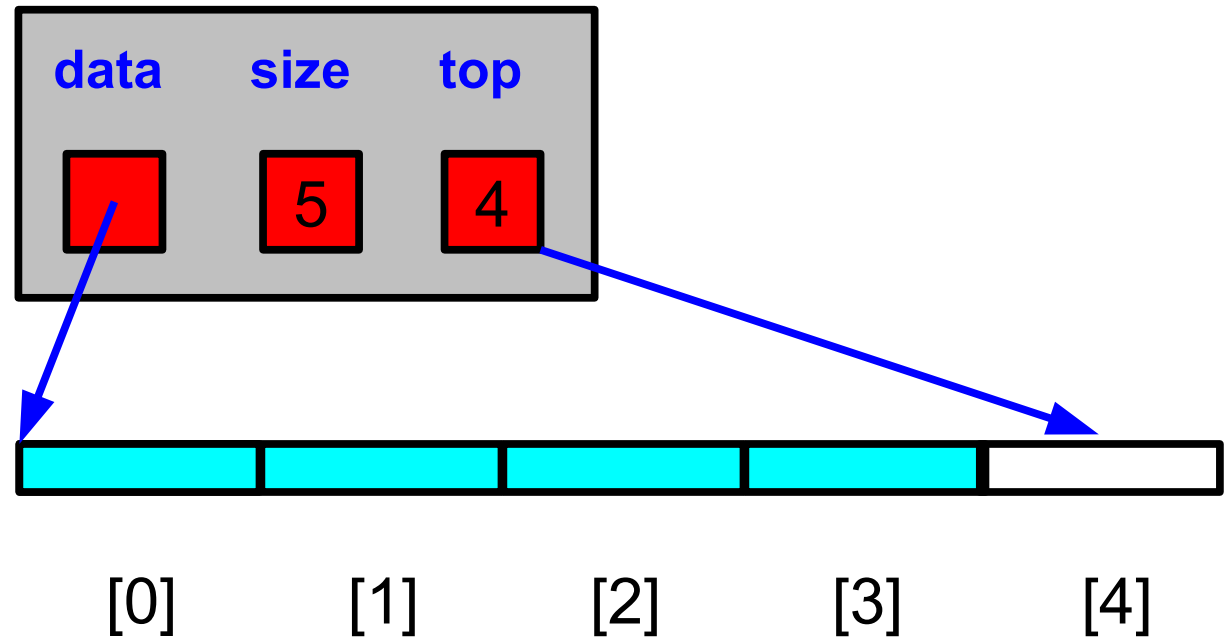
- Checks the top element. Stack is not changed

Implementing Stack with Array

- Stack is stored as an array
 - This implementation has fixed capacity



Conceptual



Physical array

Stack Implementation

```
/* stack.h */
void push(int a);
int pop(void);
int peek(void);
```

```
/* stack.c */
#include <assert.h>
#include "stack.h"

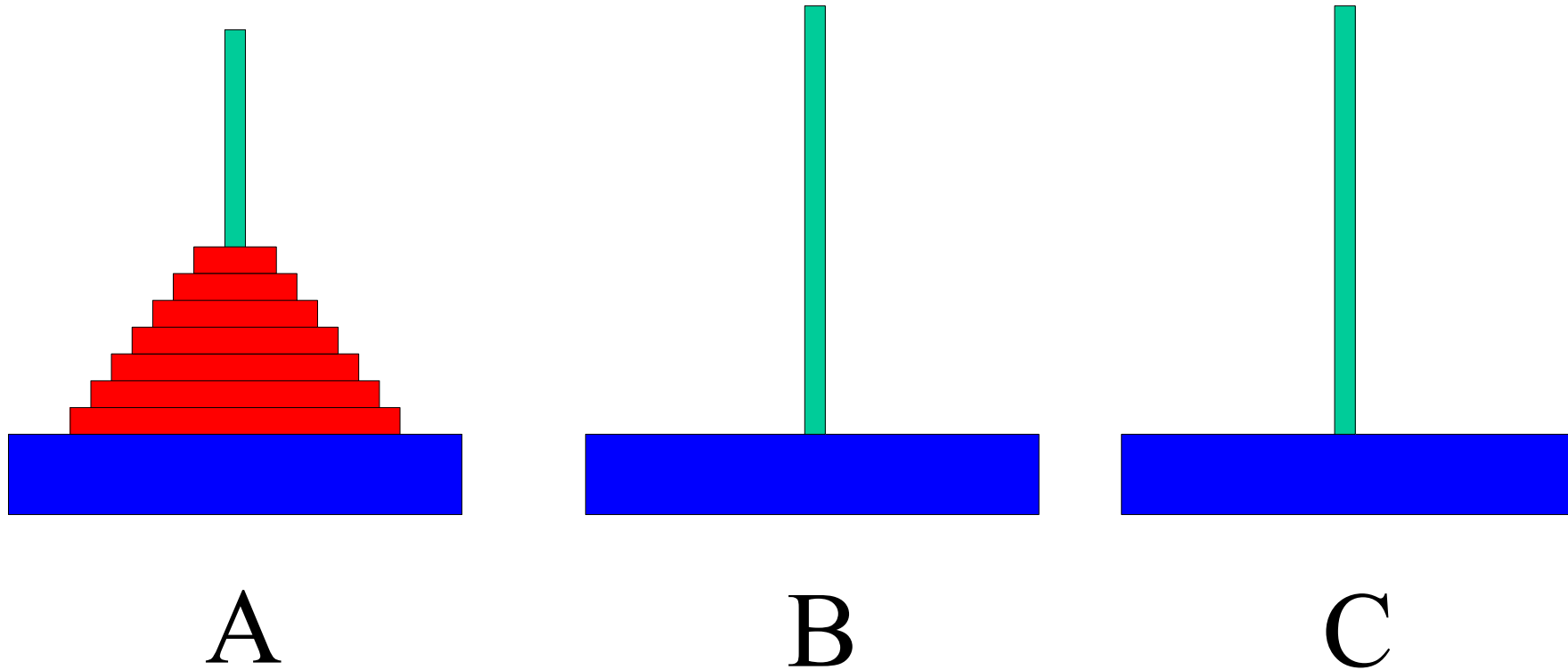
#define STACKSIZE 20
static int top; /* first empty slot on the stack */
static int size=STACKSIZE;
static int data[STACKSIZE];

void push(int a)
{
    assert(top<size);
    data[top++]=a;
}

int pop(void)
{
    assert(top>0);
    return data[--top];
}

int peek(void)
{
    assert(top>0);
    return data[top-1];
}
```

Example: Towers of Hanoi



- Object of game is to move a set of disks, stacked in successively decreasing diameters, from tower A to tower C using tower B as an intermediate; we are not allowed to stack any disk on top of a smaller disk
- Stacks can be implemented as a two-dimensional array `towers [3] [7]`

Debugging

- Frequently programs do not operate as intended
- You need some way to step through your logic other than just looking at your code
- Knowing the control flow and changes in variables as the program runs helps to find the errors
 - You can insert `printfs` in your code to print the values of variables in strategic places
 - Using a debugger is typically more convenient
- A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise
- The most popular debugger for UNIX systems is GDB, the GNU debugger

Basic Features of a Debugger

- The debugger will let you find out
 - What statement or expression did the program crash on
 - If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters
 - What are the values of program variables at a particular point during execution of the program
 - What is the result of a particular expression in a program

Using GDB

- Compile your program using `-g` option
`gcc -Wall -pedantic -g -c hello.c`
`gcc -g hello.o -o hello`
- Run the program under the control of a debugger
`gdb hello`
- Set the breakpoint at the `main` function
`break main`
- Run the program
`run`

GDB Commands

- **run *command-line-arguments***
 - Starts your program as if you had typed
hello command-line-arguments
 - You can type
run <file1 >file2
to redirect files to the standard input and output of your program
- **break *place***
 - Creates a breakpoint; the program will halt when it gets there.
The most common breakpoints are at the beginnings of functions,
as in
break main
 - You can also set breakpoints at a particular line in a source file:
break 20
break hello.c:10

GDB Commands

- **delete *N***
 - Removes breakpoint number *N*. Leave off *N* to remove all breakpoints.
 - **info break** gives info about each breakpoint
- **help *command***
 - Provides a brief description of a GDB command or topic
 - Plain **help** lists the possible topics
- **step**
 - Executes the current line of the program and stops on the next statement to be executed
- **next**
 - Like **step**, however, if the current line of the program contains a function call, it executes the function and stops at the next line

GDB Commands

- **finish**

- Continues execution until reaching the end of the current function

- **continue**

- Continues regular execution of the program until a breakpoint is hit or the program stops

- **where**

- Produces a backtrace - the chain of function calls that brought the program to its current place
- The command **backtrace** is equivalent

GDB Commands

- **print *E***
 - prints the value of *E* in the current frame in the program, where *E* is a C expression (usually just a variable)
 - **display** is similar, except every time you execute a next or step, it will print out the expression based on the new variable values
- **Quit**
 - Leave GDB

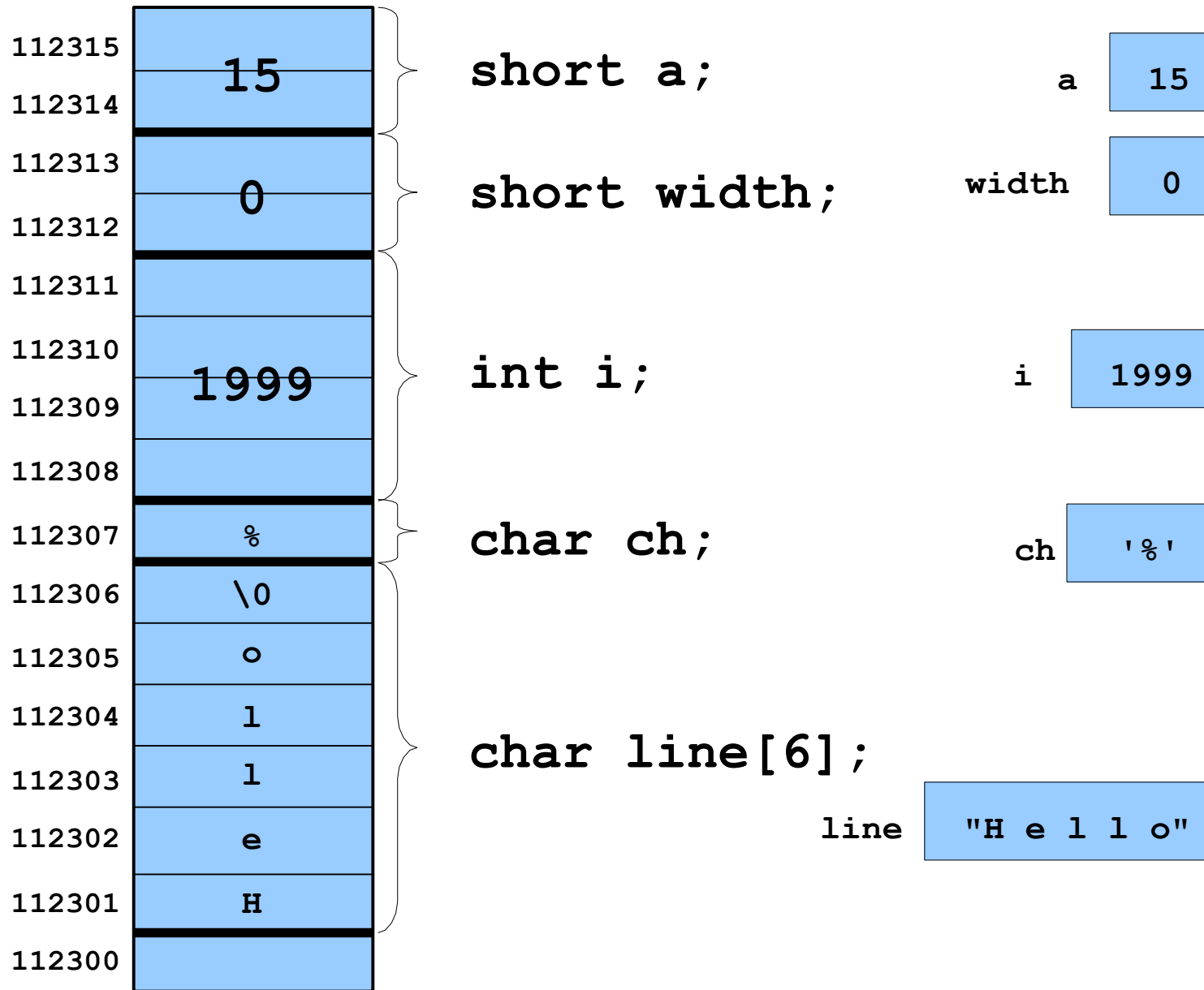
Post-Mortem Debugging

- You can make the program *dump core* when the **abort** function is invoked
- It will create a **core** file in the program directory, which contains the image of the program memory when it crashed
- You can use it to find the cause of the crash
- To enable dumping core in bash shell, type at your shell prompt
ulimit -c unlimited
- To make GDB debug the program using the specific core file use the syntax
gdb progname corefilename

Computer Memory

- Computer memory is a huge array of consecutively numbered memory cells
- A cell can store one byte of data
- An address of a memory cell is its number in an array
- Data of different type occupies one or more contiguous cells (bytes)

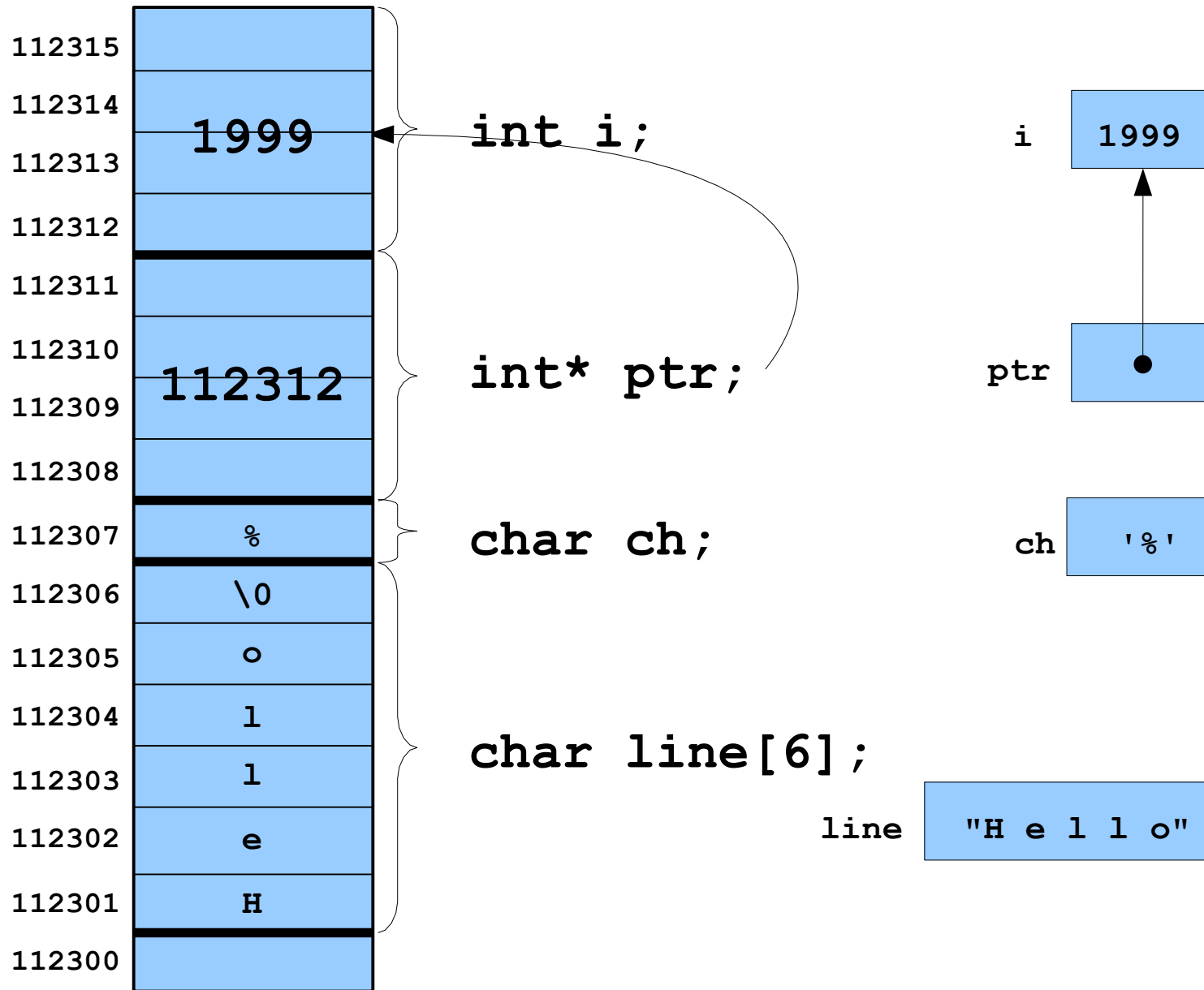
Computer Memory



Pointers

- A pointer is a variable that can store an address of a variable (i.e., 112300)
- We say that a pointer points to a variable that is stored at that address
- A pointer itself usually occupies 4 bytes of memory (then it can address cells from 0 to $2^{32}-1$)

Computer Memory



Declaring Pointers

- In C you can specify the type of variable a pointer can point to:

```
int *ad; /* pointer to int */
```

```
char *s; /* pointer to char */
```

```
float *fp; /* pointer to float */
```

```
char **s; /* pointer to variable that is a  
pointer to char */
```

Operations with Pointers

- We can assign an address to a pointer

```
int *p1, *p2; int a, b;
```

```
p1 = &a;
```

- We can assign pointers to each other

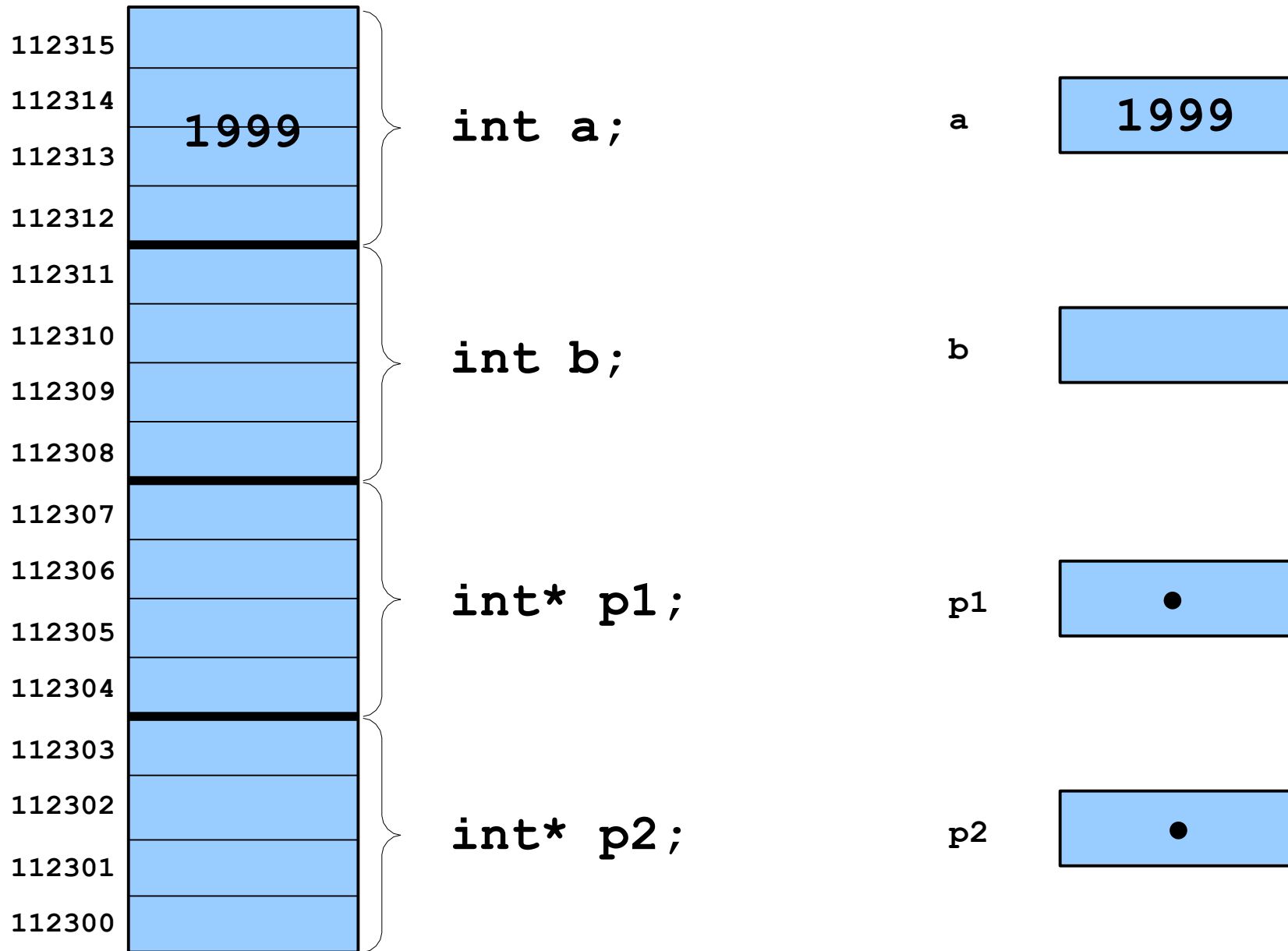
```
p2 = p1;
```

- We can dereference pointers

```
*p1 = 3; /* same as a = 3; */
```

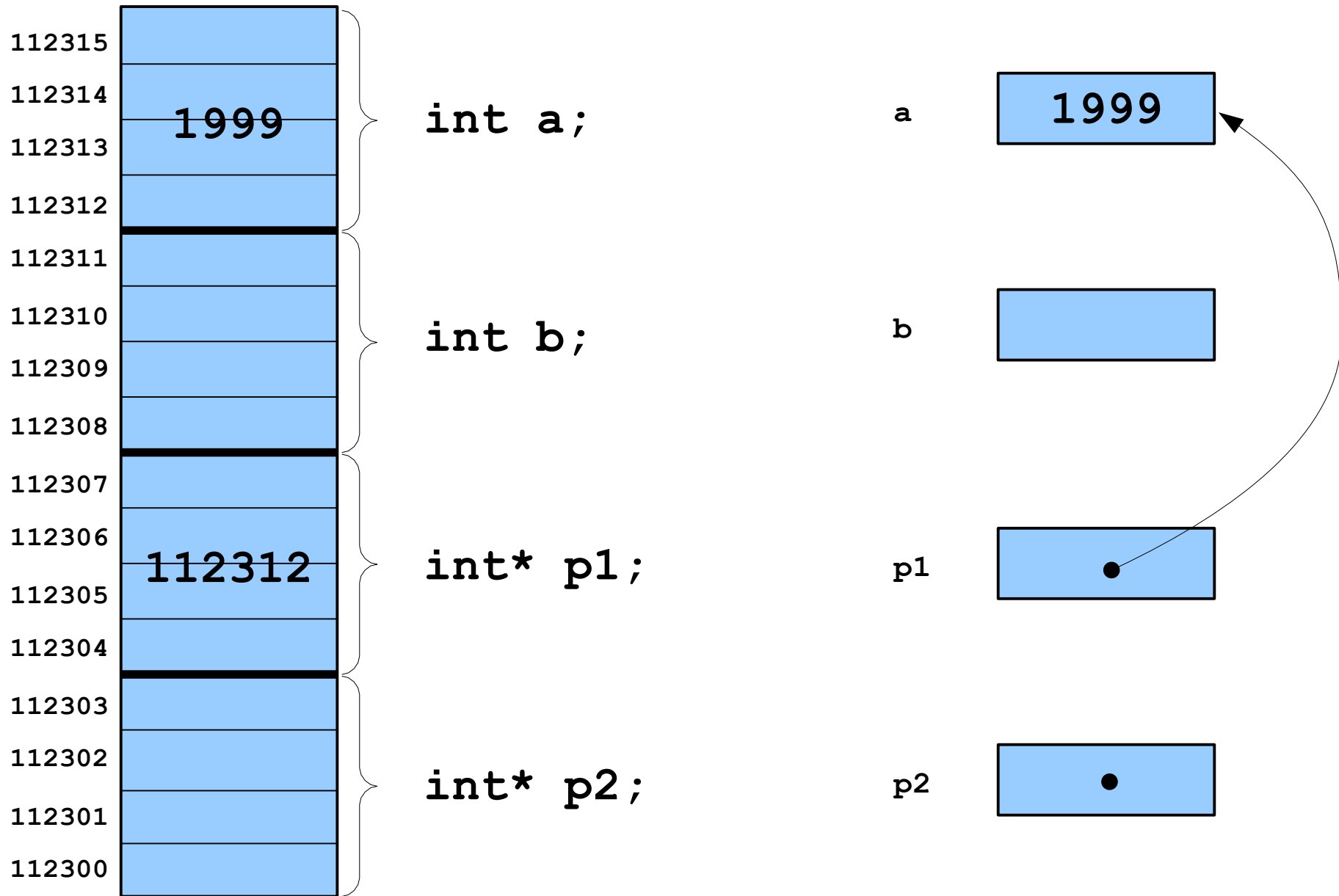
```
b = *p1; /* same as b = a; */
```

Operations & and *



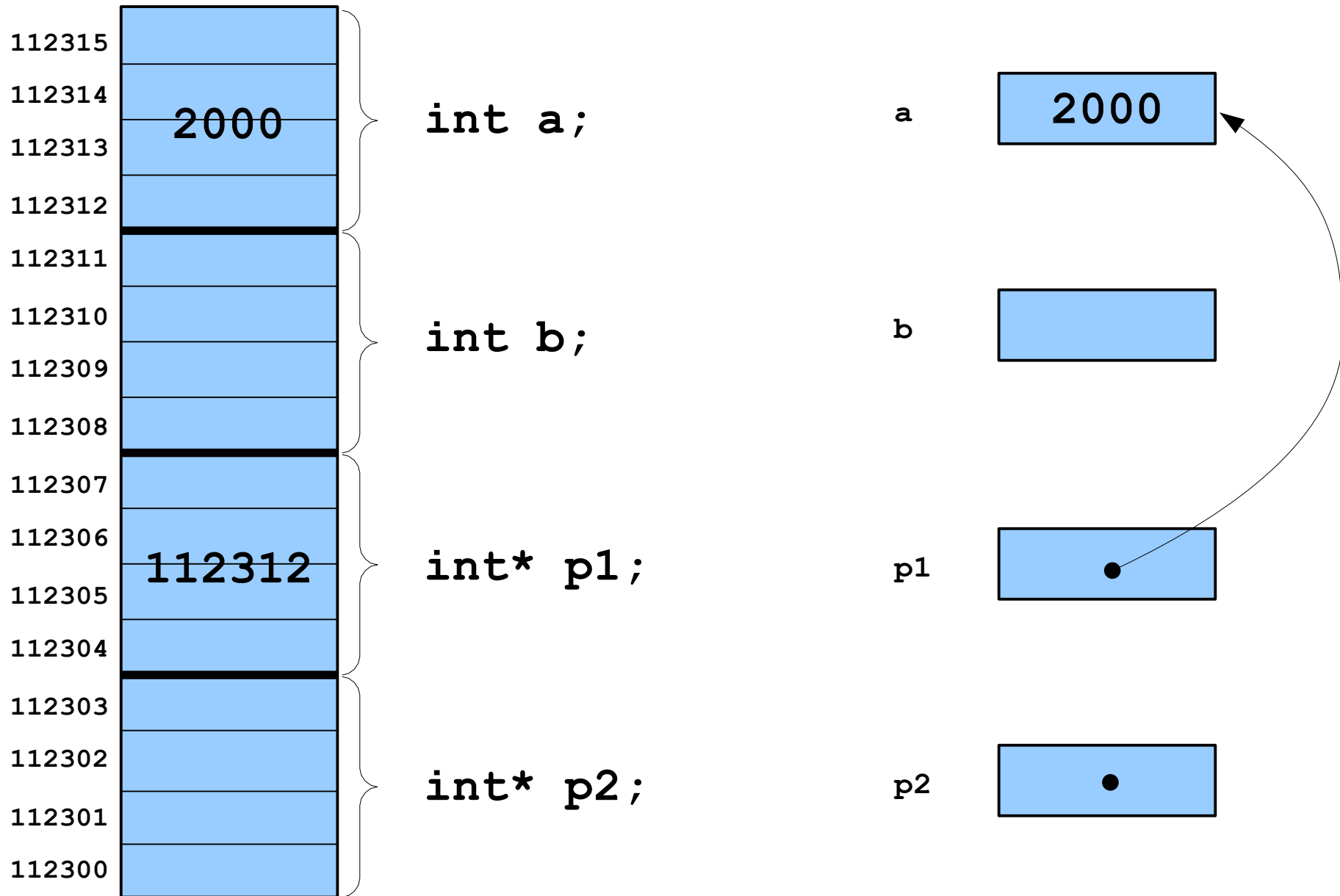
`a = 1999;`

Operations & and *



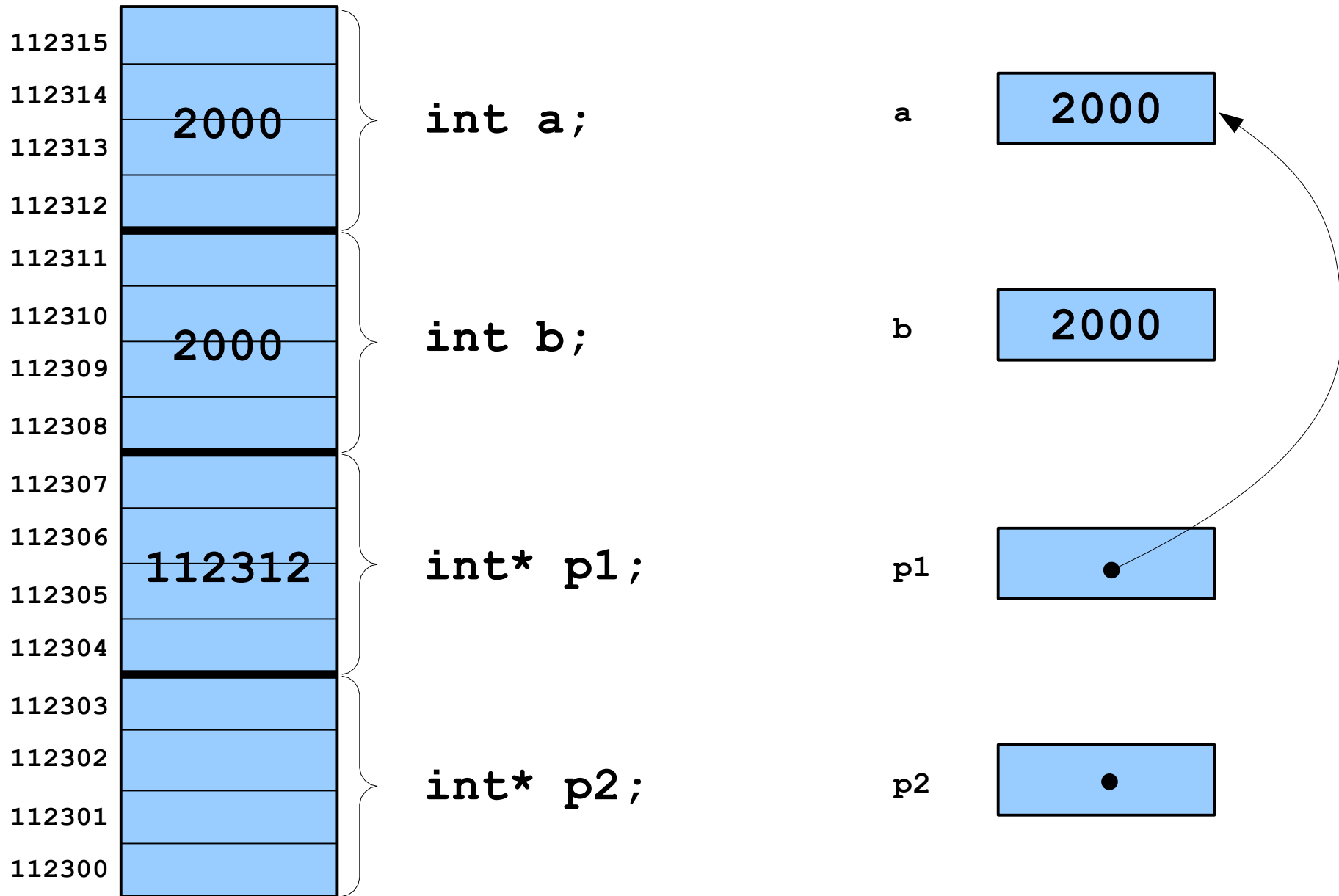
`p1 = &a;`

Operations & and *



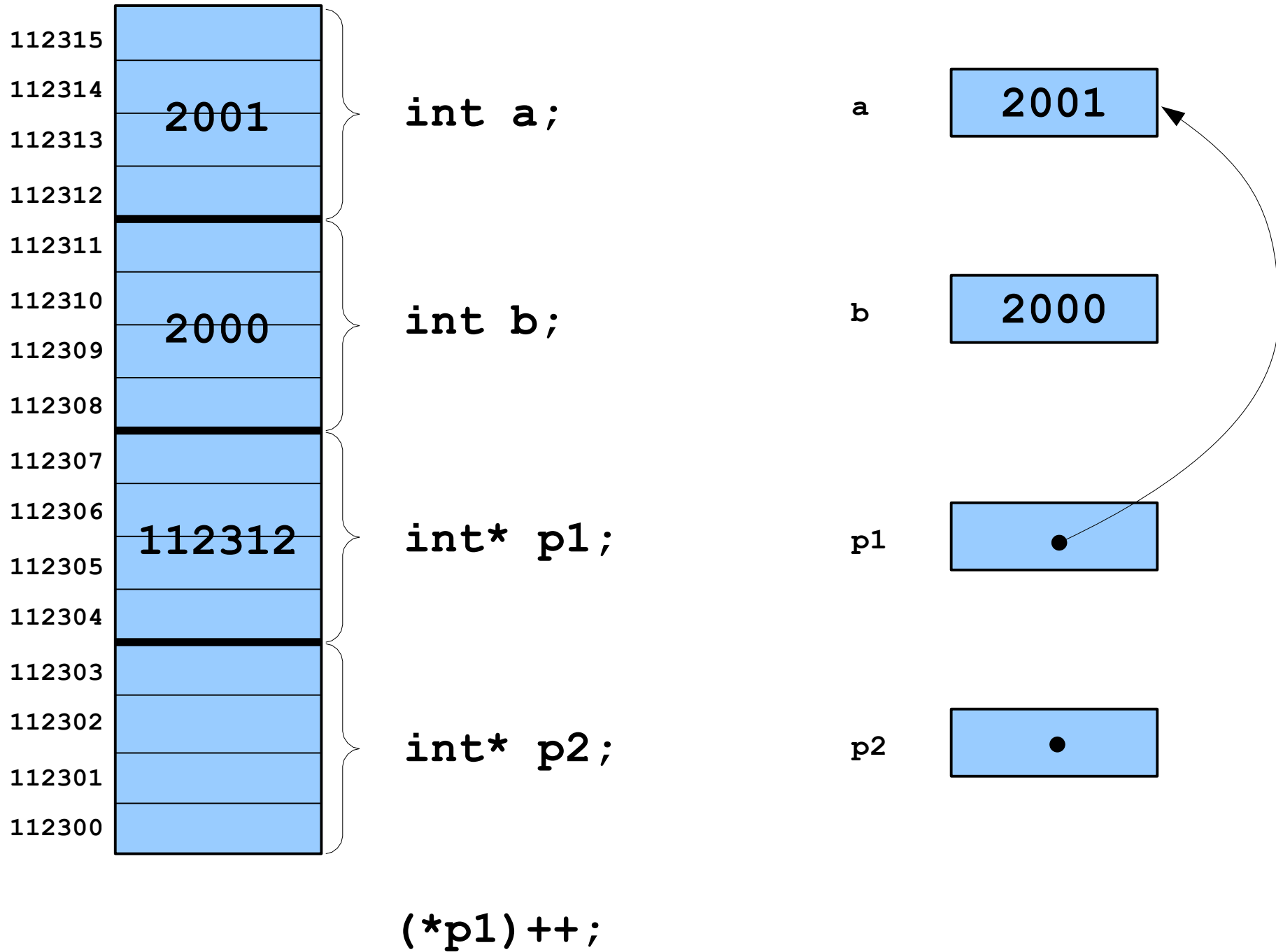
`*p1 = 2000;`

Operations & and *

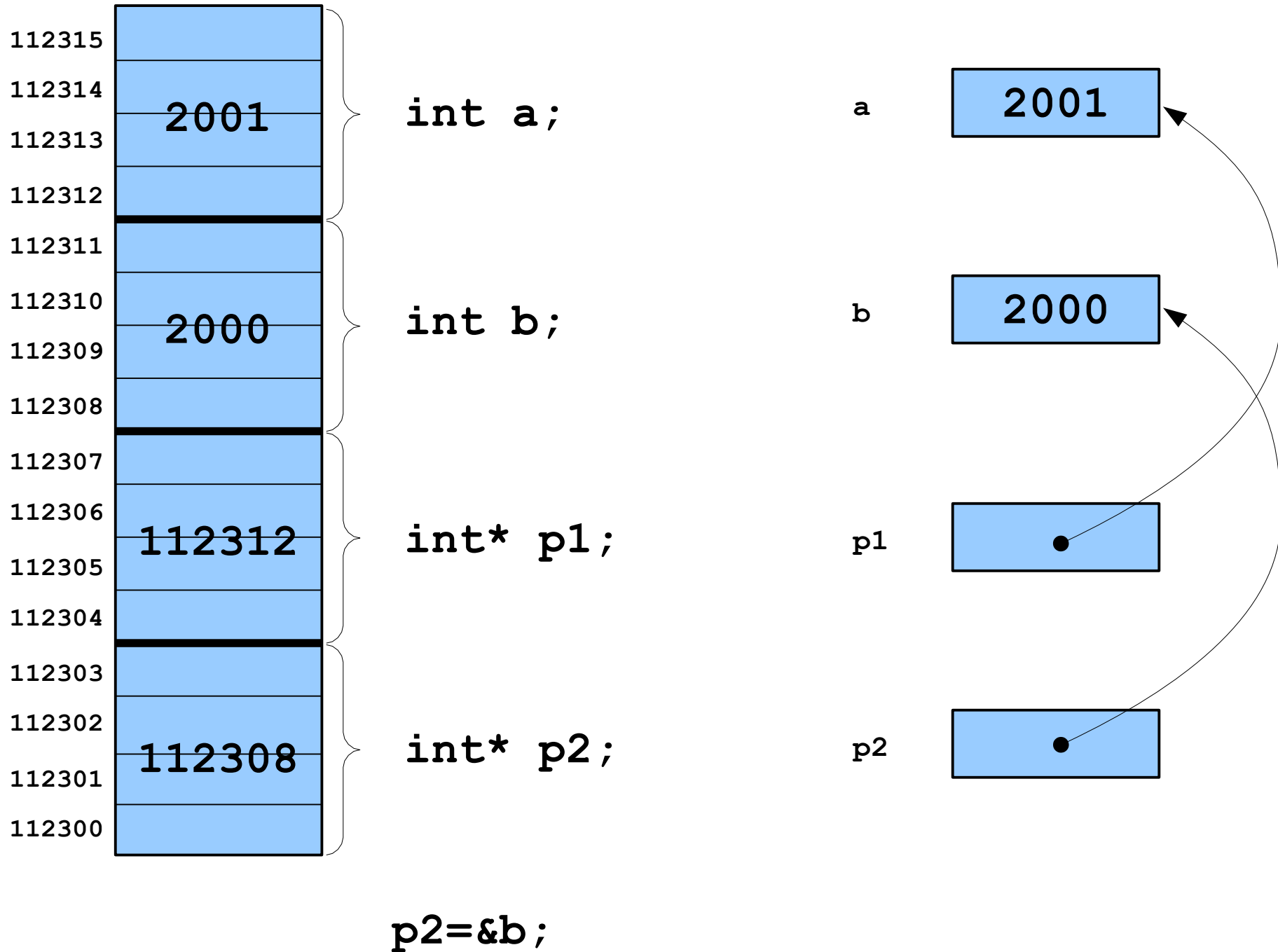


`b=*p1;`

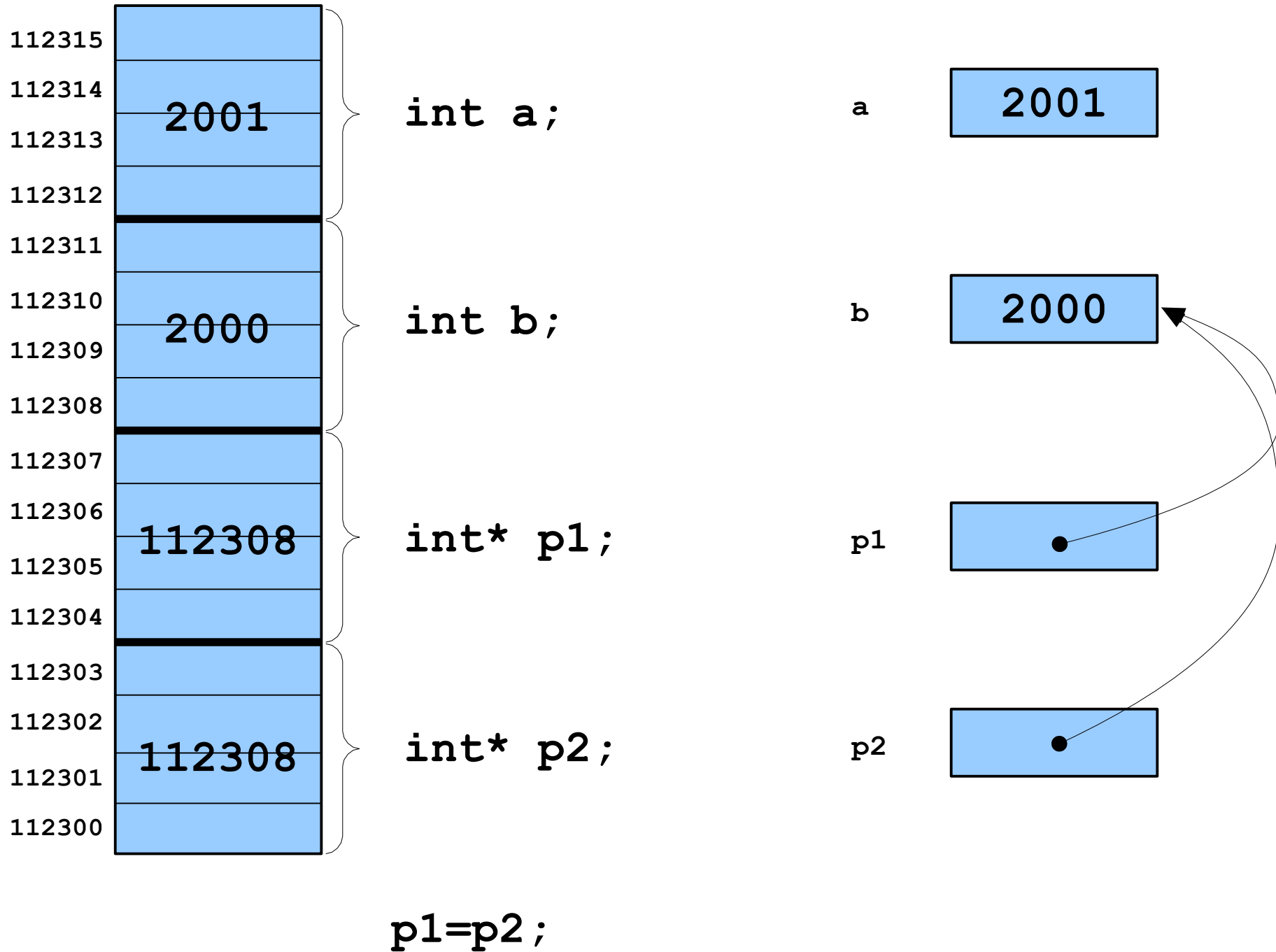
Operations & and *



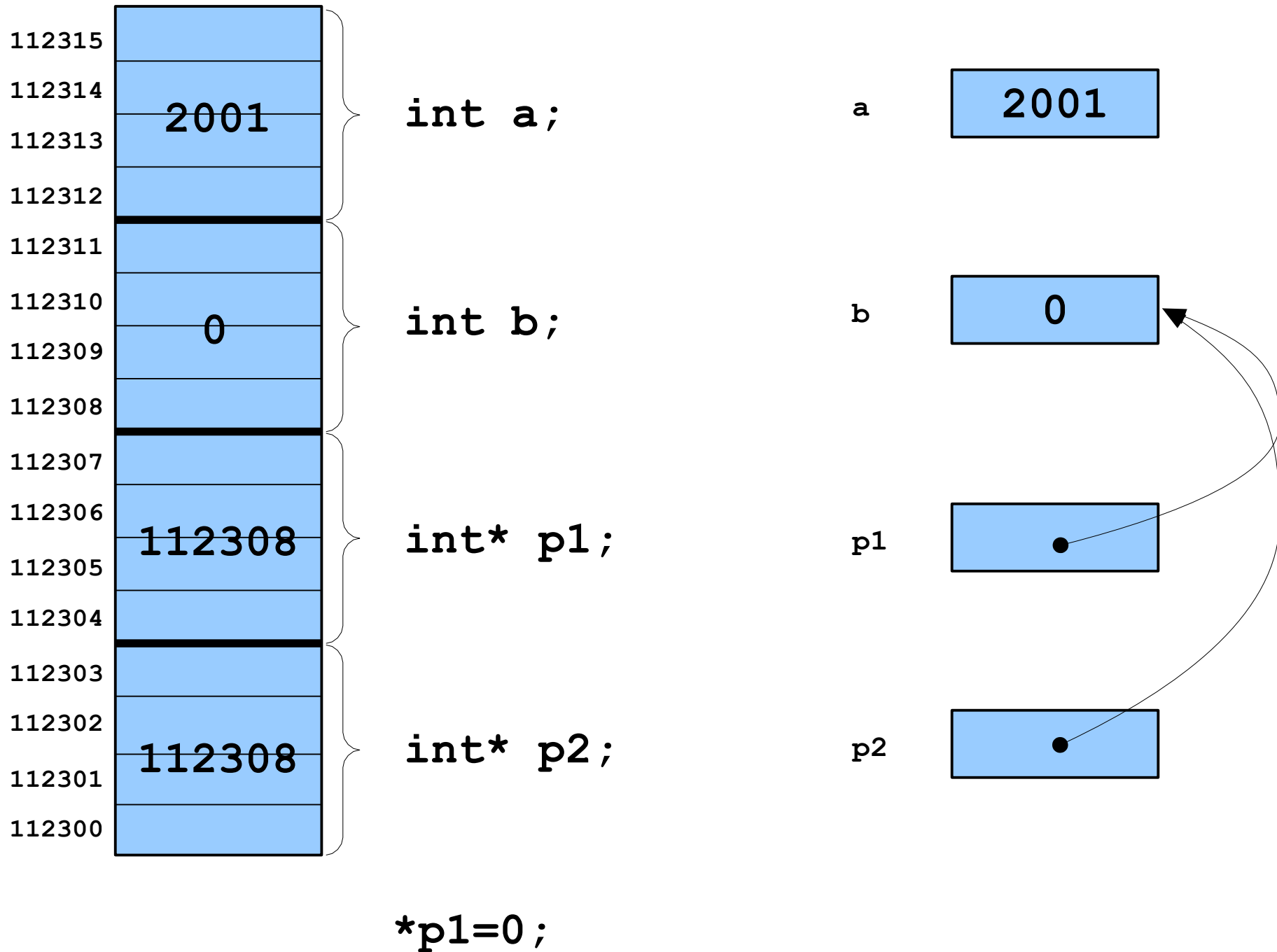
Operations & and *



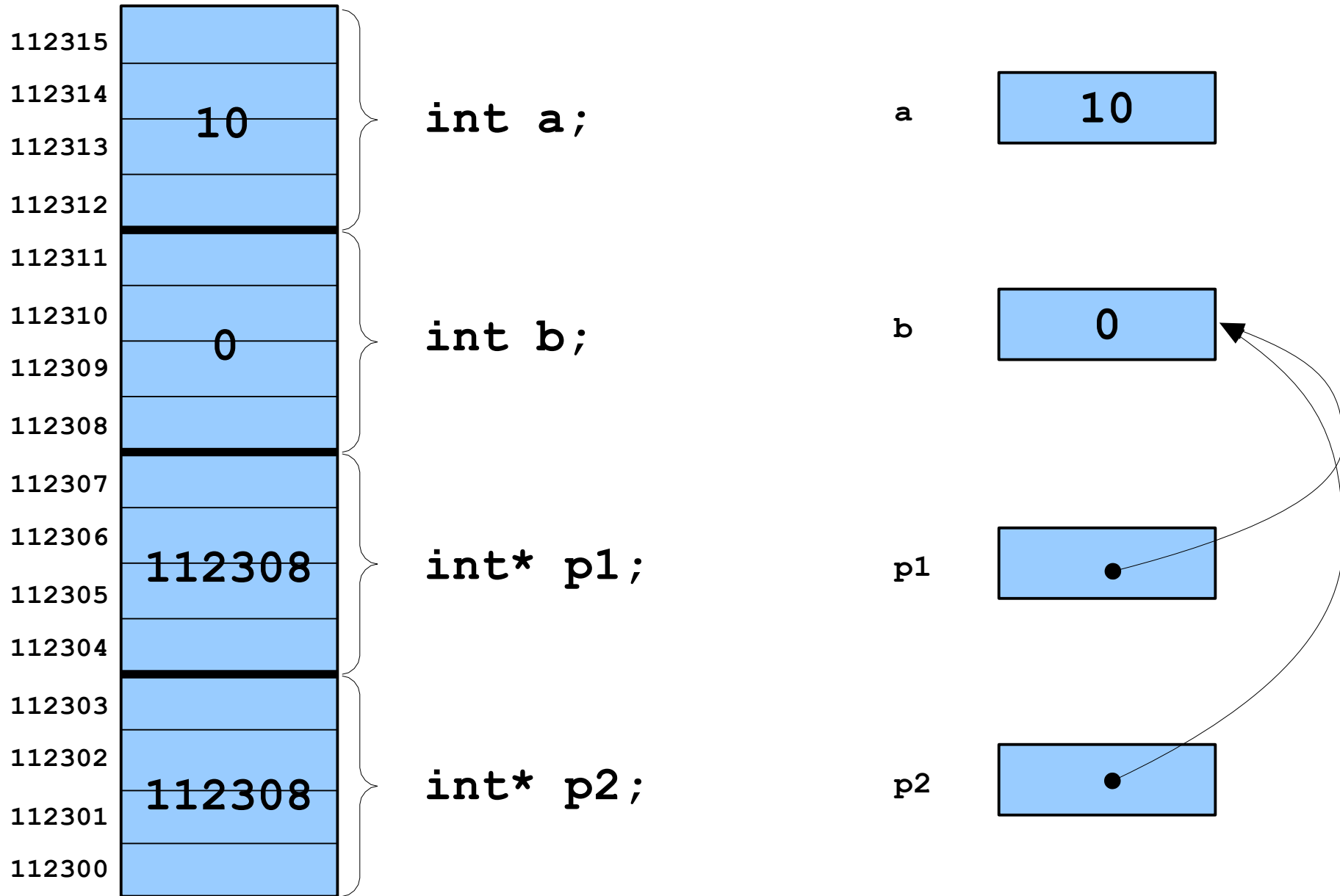
Operations & and *



Operations & and *



Operations & and *



`a=*p2+10;`

Pointer Function Arguments

- Function arguments in C are strictly pass-by-value
- However, we can simulate pass-by-reference by passing a pointer
- This is very useful when you need to
 - Support in/out (bi-directional) parameters (e.g. swap, find-replace)
 - Return multiple outputs (one return value isn't enough)
 - Pass around large objects (arrays and structures)

Pointers and Function Arguments

```
/* bad example of swapping
a function can't change parameters */
void bad_swap(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

```
/* good example of swapping - a function can't change parameters,
   but if a parameter is a pointer it can change the value it points to */
void good_swap(int *px, int *py)
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Pointers and Function Arguments

```
#include <stdio.h>

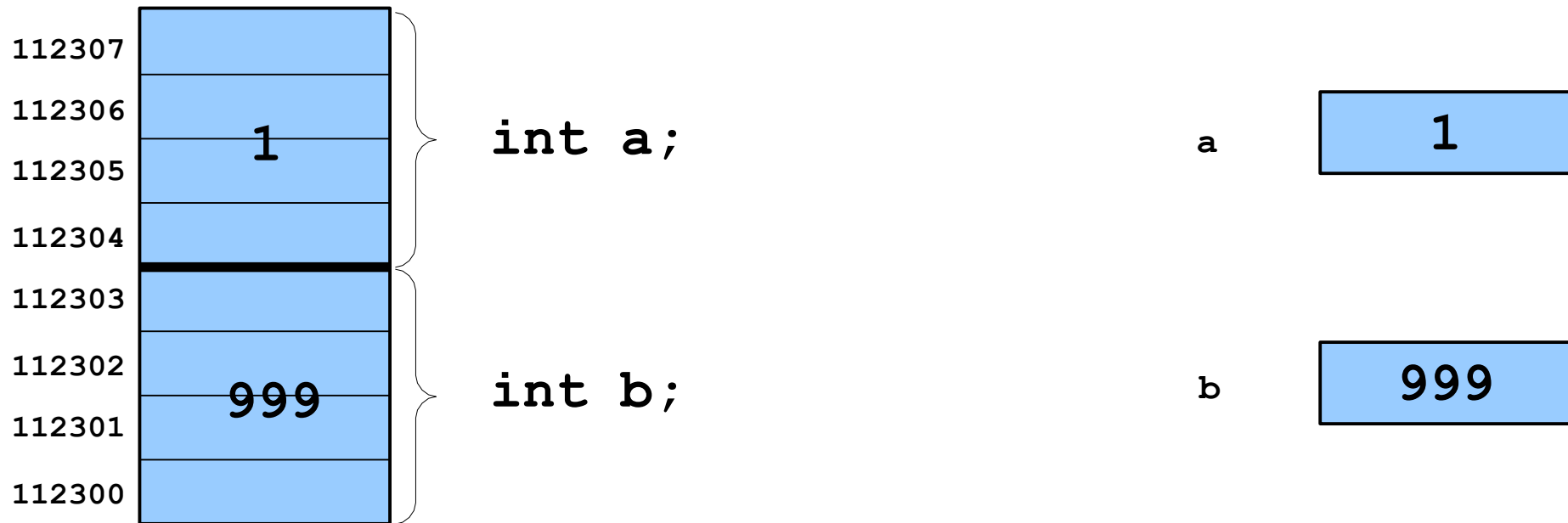
void bad_swap(int x, int y);
void good_swap(int *p1, int *p2);

main() {
    int a = 1, b = 999;

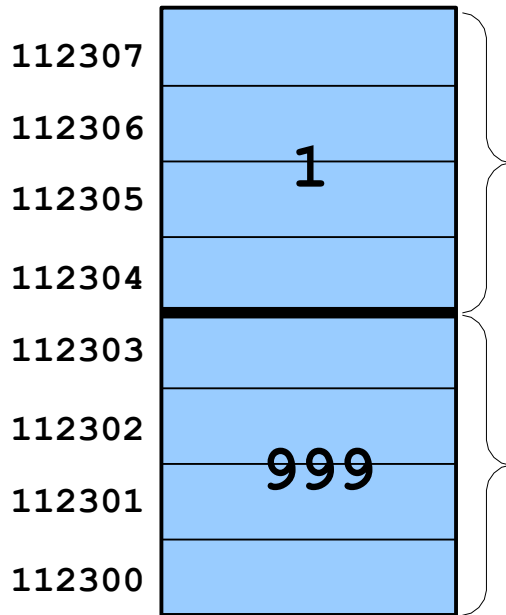
    printf("a = %d, b = %d\n", a, b);
    bad_swap(a, b);
    printf("a = %d, b = %d\n", a, b);
    good_swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
}
```


Inside Story

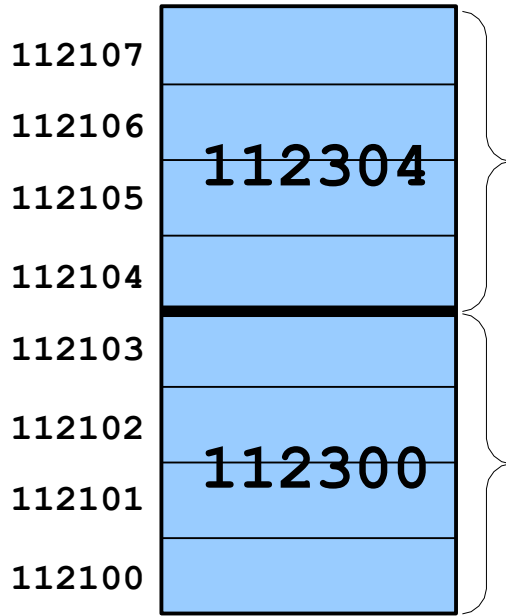
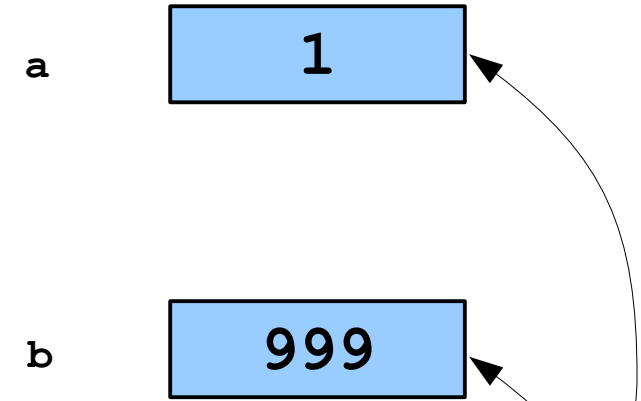
`main`



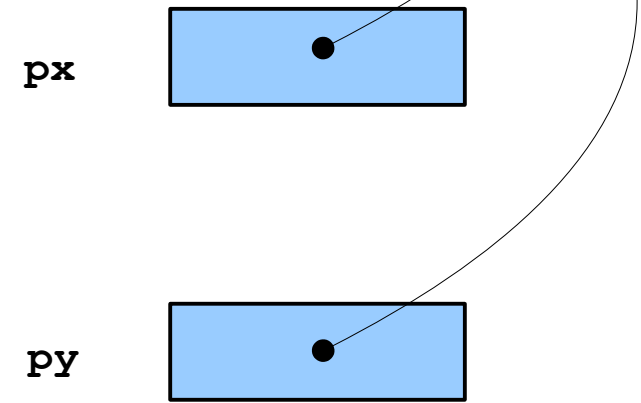
Inside Story



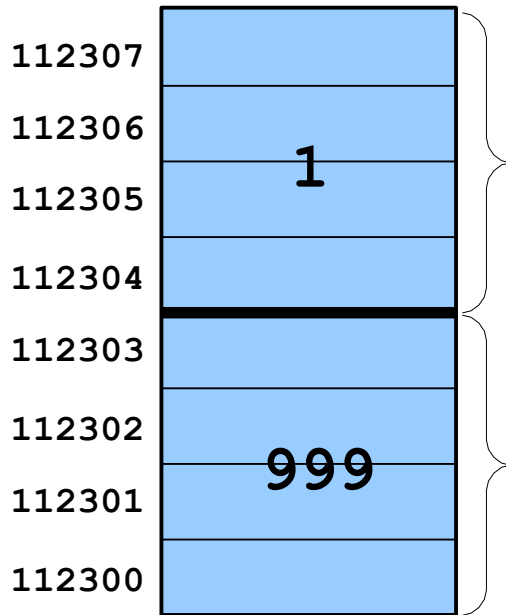
```
main  
  
int a;  
  
int b;  
  
void swap(int *px,  
          int *py)  
{  
    int temp;  
  
    temp = *px;  
    *px = *py;  
    *py = temp;  
}
```



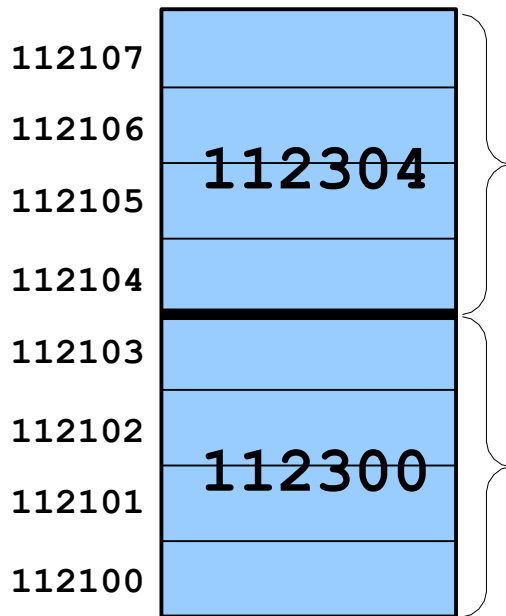
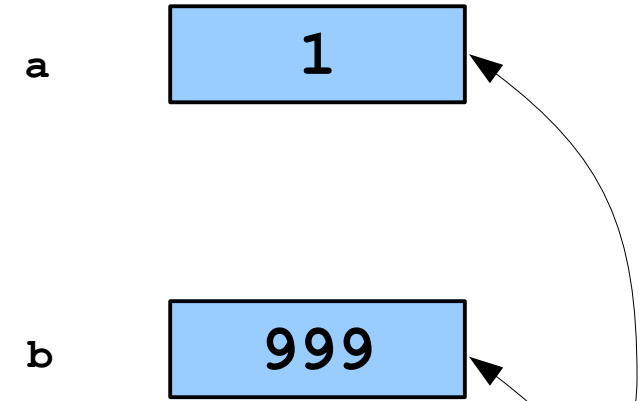
```
swap  
  
int* px;  
  
int* py;  
  
swap (&a, &b);
```



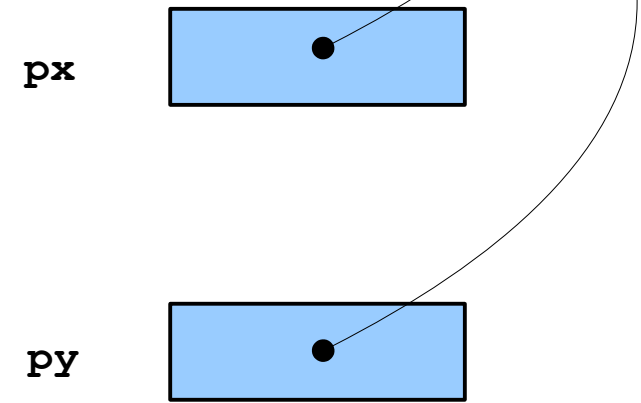
Inside Story



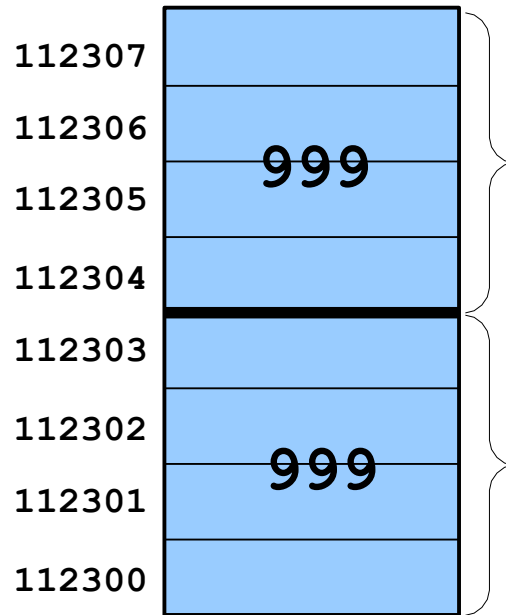
```
main
int a;
void swap(int *px,
          int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int b;
```



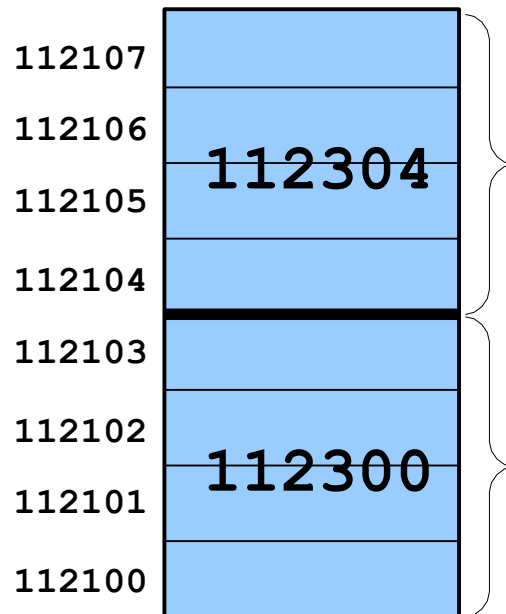
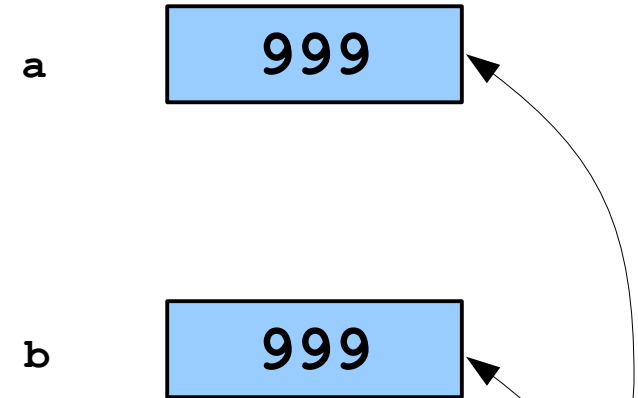
```
swap
int* px;
temp 1
int* py;
temp=*px;
```



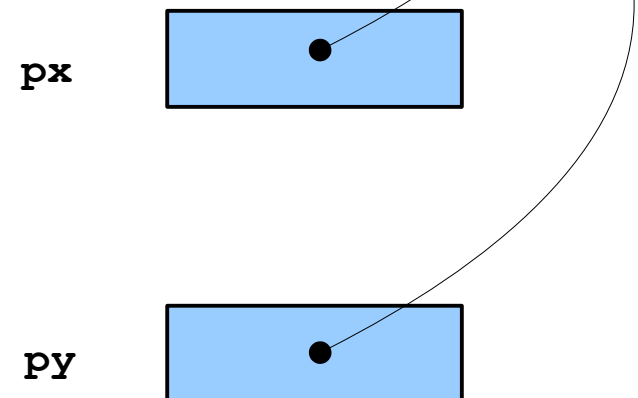
Inside Story



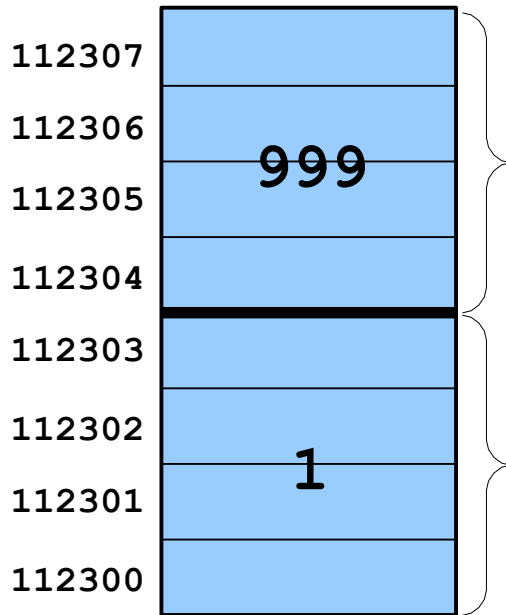
```
main
int a;
void swap(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int b;
```



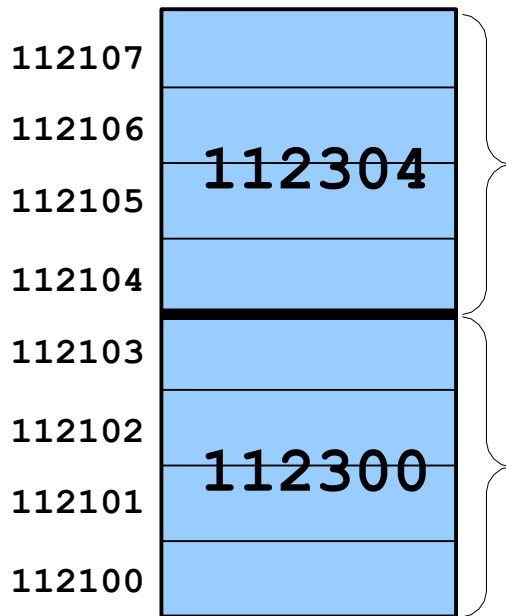
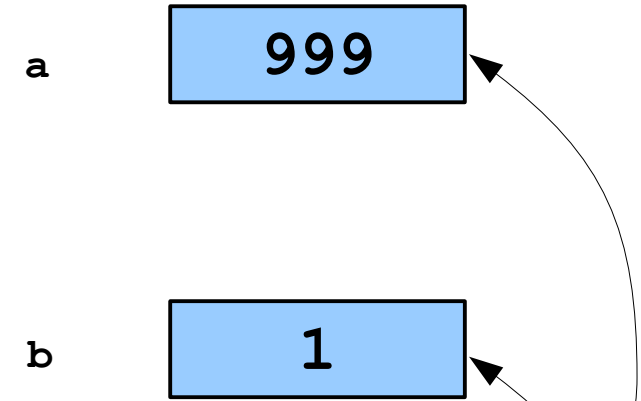
```
swap
int* px;
temp 1
int* py;
*px=*py;
```



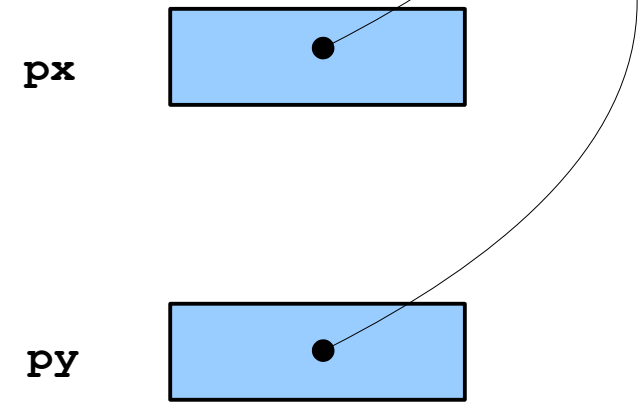
Inside Story



```
main
int a;
void swap(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int b;
```

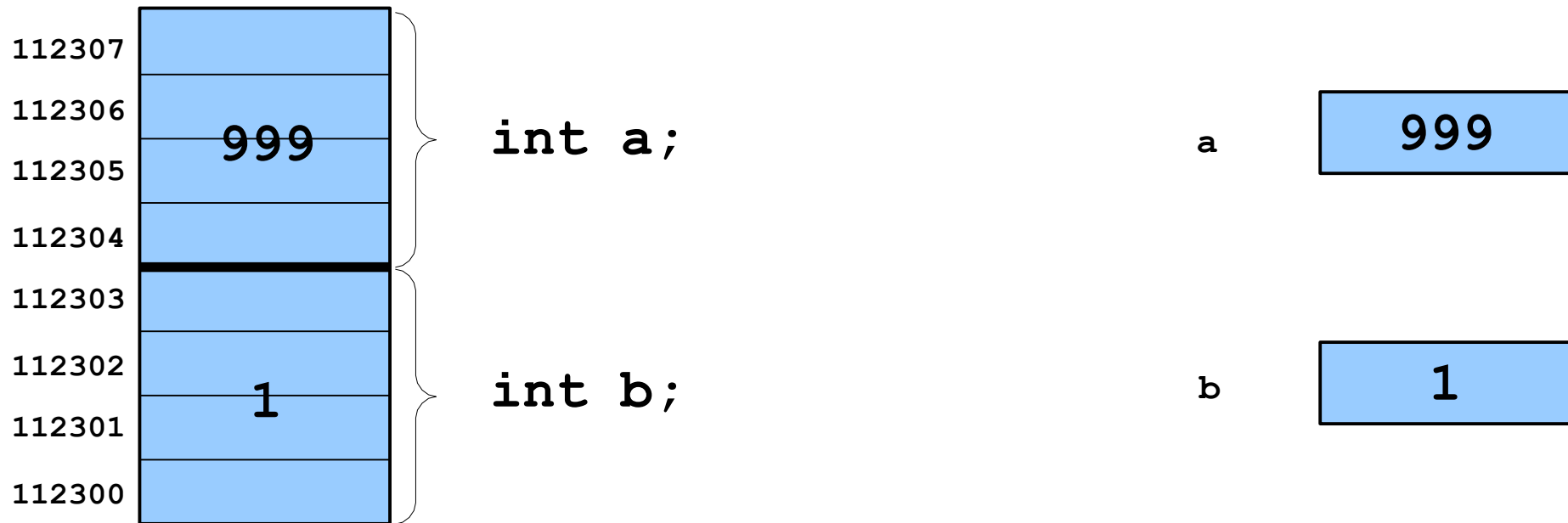


```
swap
int* px;
temp 1
int* py;
*py=temp;
```



Inside Story

`main`



Pointer Function Arguments

```
#include <stdio.h>    /* printf() */
#include <math.h>     /* pow() */

static void CalcAreaVol(double dRad, double* pdArea,
                        double* pdVol)
{
    *pdArea = 4.0 * M_PI * pow(dRad, 2.0);
    *pdVol  = 4.0/3.0 * M_PI * pow(dRad, 3.0);
}

int main()
{
    double area = 0.0, vol = 0.0;
    CalcAreaVol(3.5, &area, &vol);
    printf("Radius=%g Area=%g Vol=%g\n", 3.5,
           area, vol);
    return 0;
}
```

Reading Input - scanf

- If we need to read a value from the standard input into a variable we can use `scanf`
- `scanf` is able to put the input into a variable since we pass its address

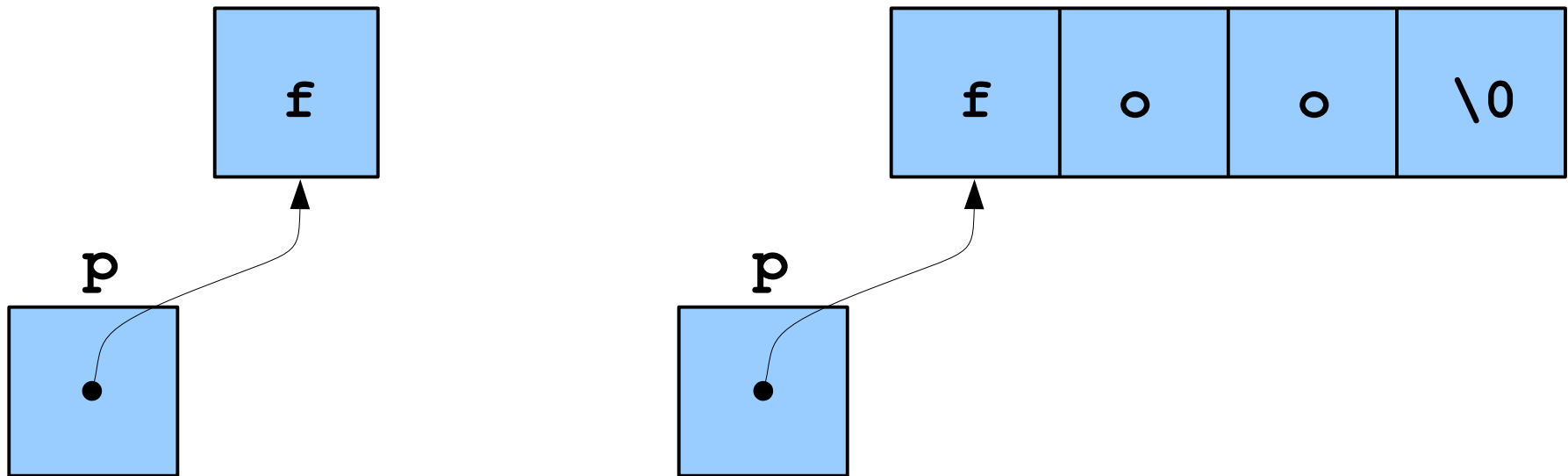
```
int capital = 0;  
scanf("%d", &capital);
```

- What happens without `&`?

```
scanf("%d", capital);
```
- `scanf` thinks that the value of `capital` is an address where it has to place data!

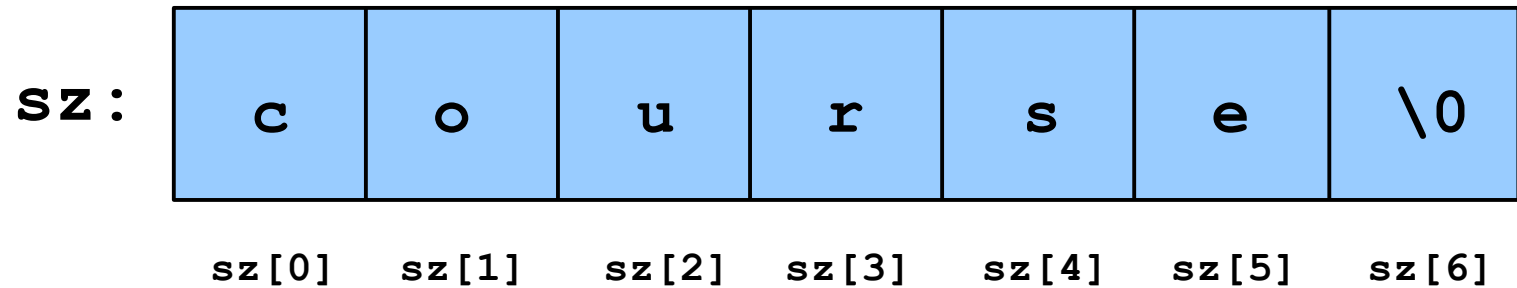
Pointers to Arrays

- A pointer can point to
 - a single value (`int`, `char`, `float`, `double`, `struct`, etc.)
 - an array of values
- Can be confusing
- Consider: `char* p`
- Need to know if its pointing to a single char or to the first element in an array of chars?

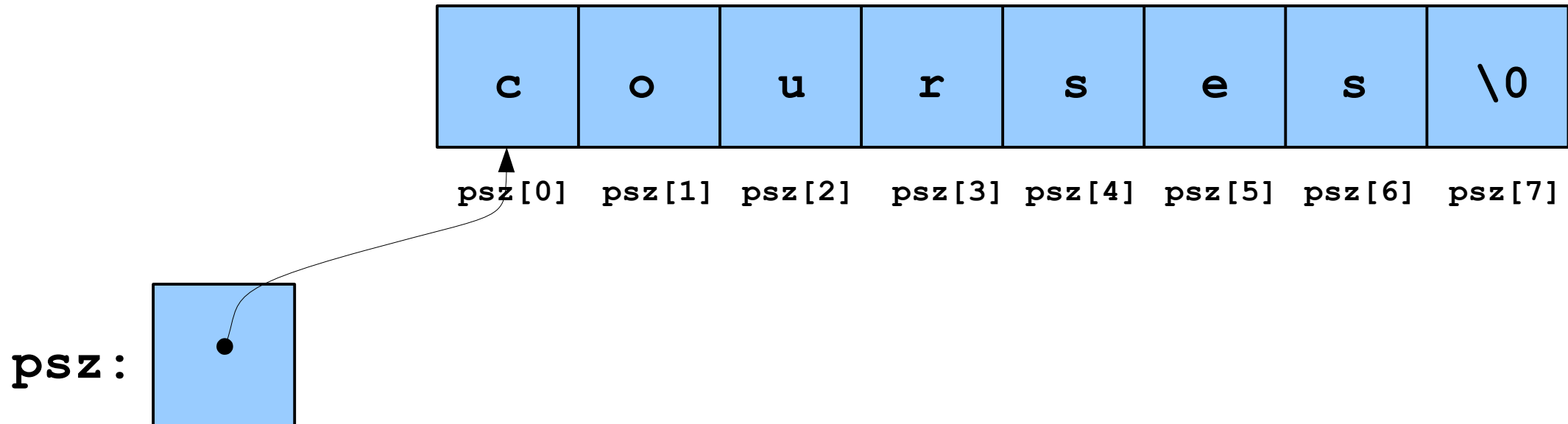


Strings Revisited

```
char sz[] = "course";
```



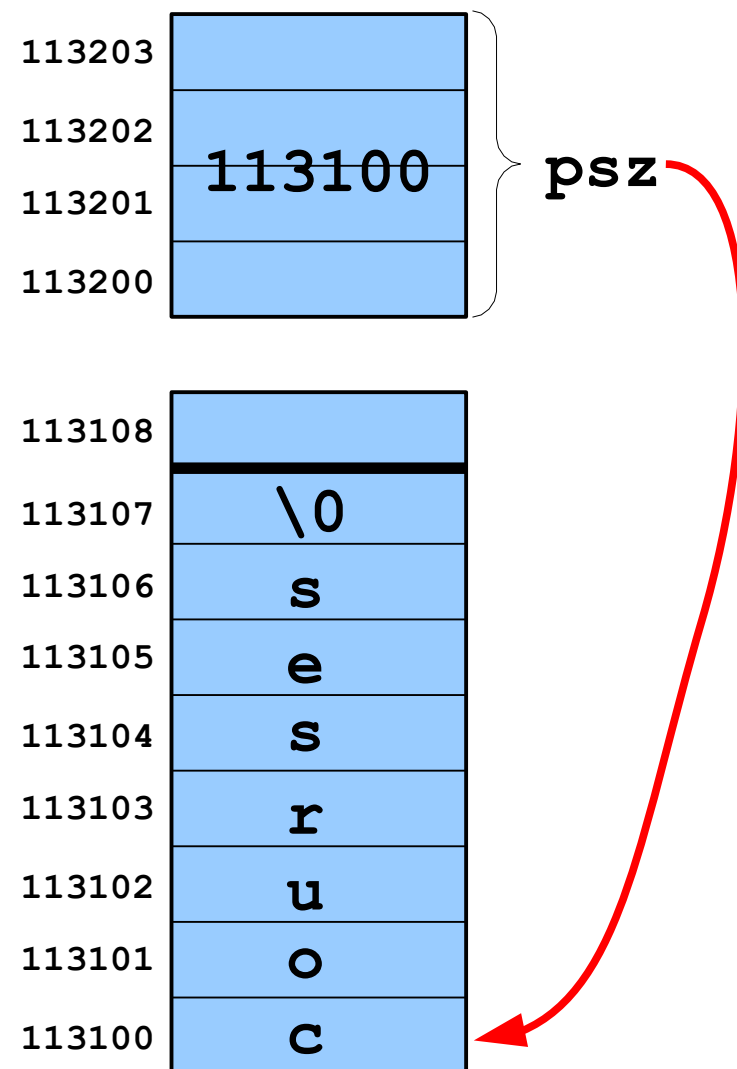
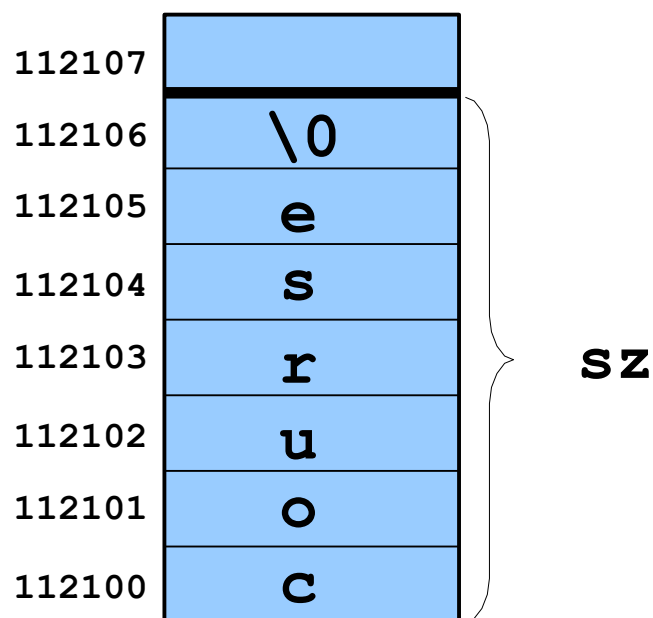
```
char* psz = "courses";
```



Strings Revisited

```
char sz[] = "course";  
char* psz = "courses";
```

- **sz** is an array of characters
- **psz** is a pointer to an array of characters

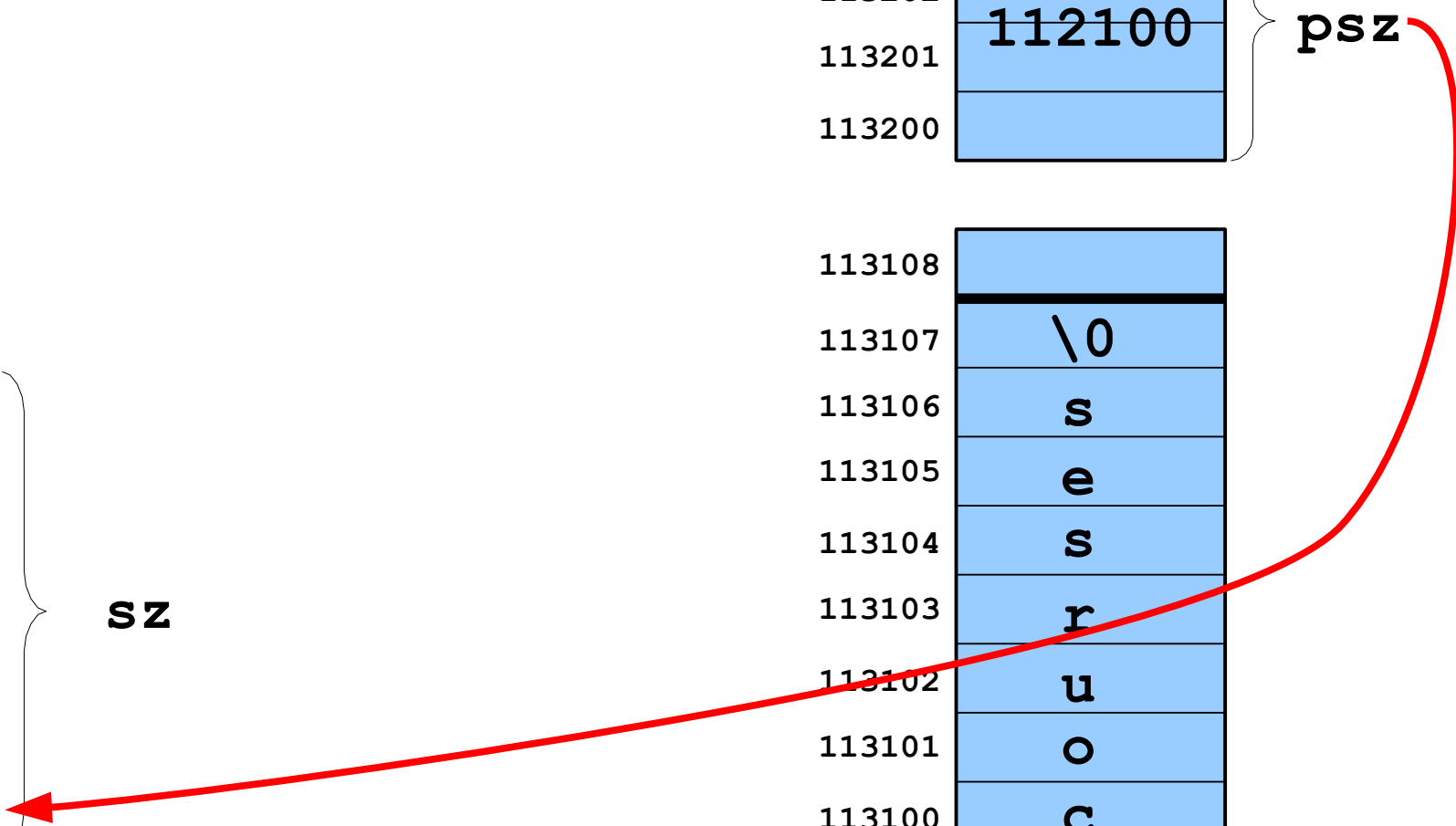
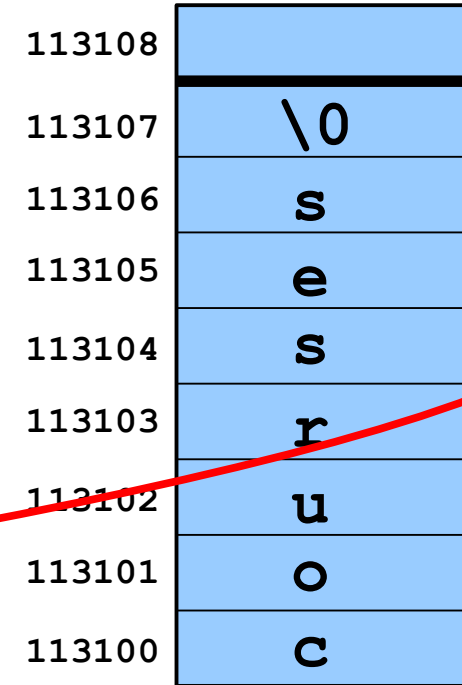
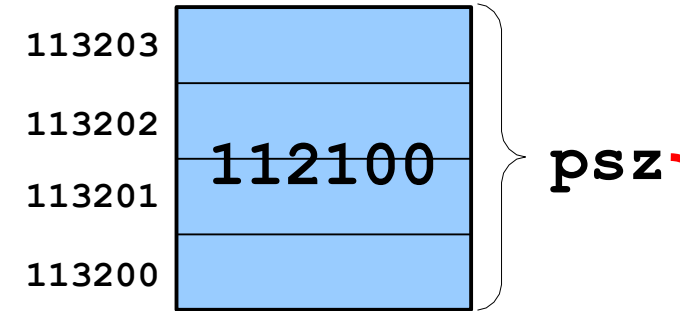
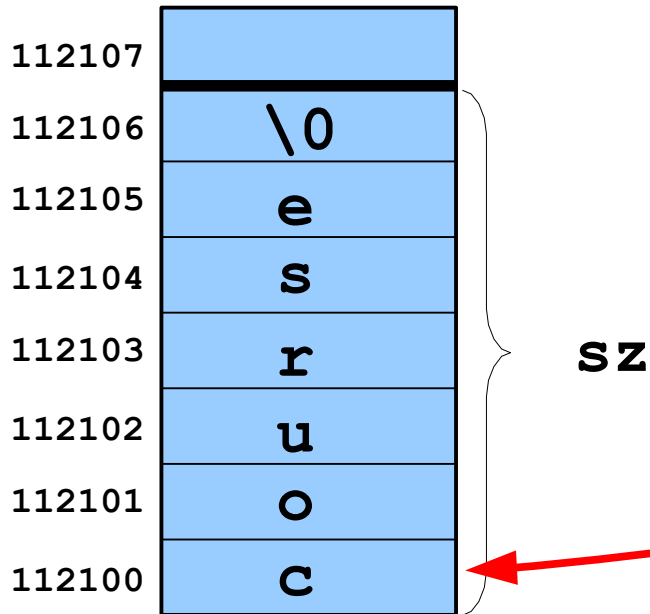


- **sizeof(sz)** ? **sizeof(psz)** ?

Strings Revisited

- You can change the pointer to point to something else

```
psz = sz;
```



Strings Revisited

- You can modify characters in `sz`

```
sz[0] = 'C'; /* Okay */
```

- You cannot modify characters in `psz`

```
psz[0] = 'C'; /* Danger, Will Robinson!
```

- `psz` is immutable (read-only). Why?

- It points to a string literal ("courses") that usually resides in read-only memory

- `psz[0] = 'C'` is undefined (implementation dependent)

- On Linux, this causes a segmentation violation

- `const` qualifier is used to prevent accidental modification

```
const char* psz = "courses";
```

- Also might use `const` for `sz`

```
const char sz[] = "course";
```



[] and * Similarities

- From `stdio.h`:

```
size_t strlen(const char*);
```

- Can pass an array wherever a pointer is expected

```
strlen(sz);
```

```
strlen(psz);
```

```
strlen("Greetings!");
```

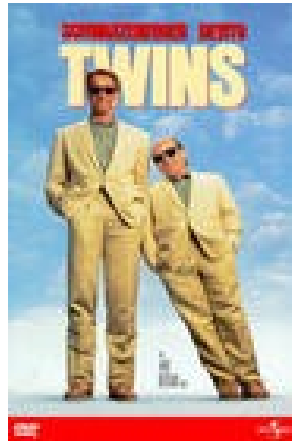
- Equivalent declaration:

```
size_t strlen(const char[]);
```

- Array names are implicitly converted to a pointer to their first element

```
strlen(sz);      ⇒ strlen(&sz[0]);
```

```
psz = sz;       ⇒ psz = &sz[0];
```



[] and * Similarities

- Only two things can be done to an array
 - Determine its size using `sizeof()`
 - Obtain a pointer to the first element
- All other array operations are actually done with pointers
- Subscripts are automatically converted to pointer operations

Pointer Arithmetic

- Can use subscript notation to access elements: `p[i]`

```
char sz[] = "course";
```

```
char* psz = sz;
```

```
sz[0] = 'C';           ⇒ "Course"
```

```
psz[5] = 'E';         ⇒ "CourseE"
```

- Can also use pointer arithmetic: `*(p + I)`

```
*(sz + 0) = 'C';
```

```
*(psz + 5) = 'E';
```


Pointer Arithmetic

- This works also with larger types

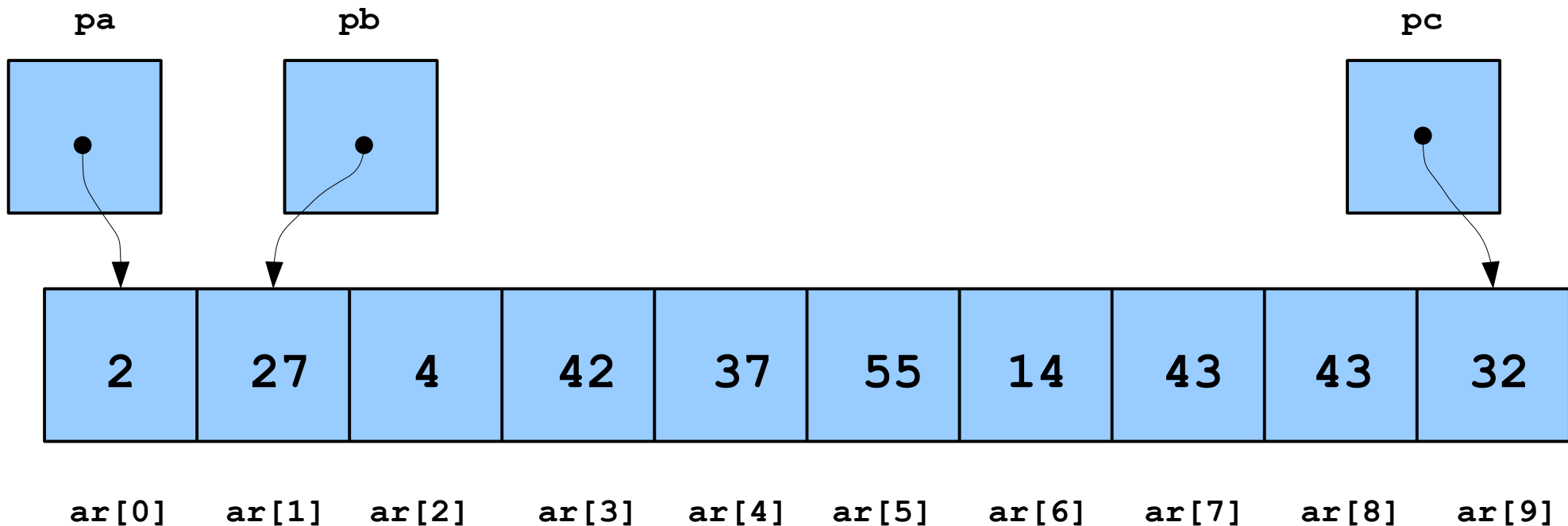
```
int ar[10];
```

```
int *pa, *pb, *pc;
```

```
pa = &ar[0] /* same as pa = ar */
```

```
pb = pa + 1; pc = pa + 9;
```

```
ar[3]=42; *(ar+4)=37; *pa=2; *pb=27; *pc=32;
```



Pointer Arithmetic

- We can add and subtract integers to/from pointers - the result is a pointer to another element of this type

```
int *pa; char *s;
```

s-1 ⇒ points to **char** before **s** (1 subtracted)

pa+1 ⇒ points to next **int** (4 added!)

s+9 ⇒ points to 9th **char** after **s** (9 added)

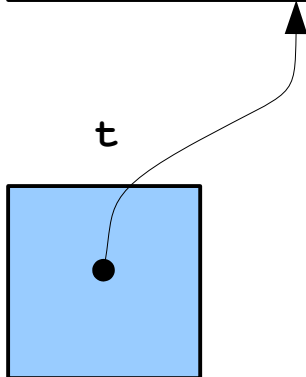
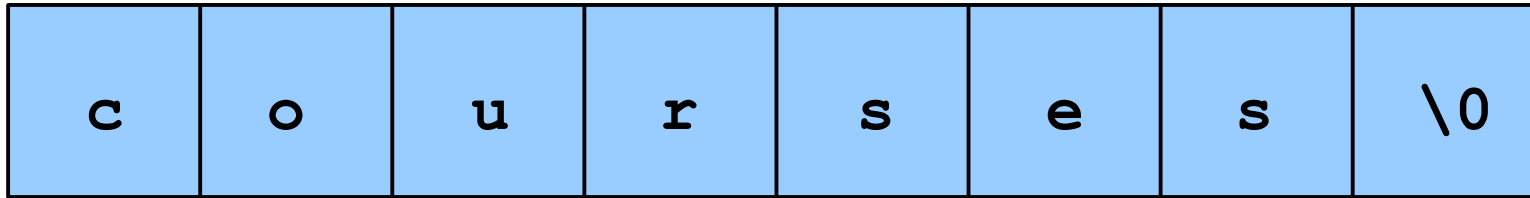
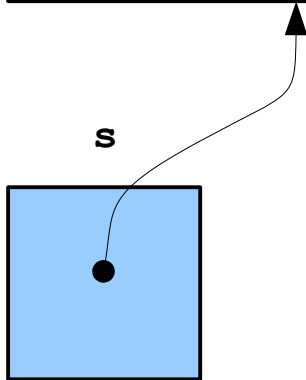
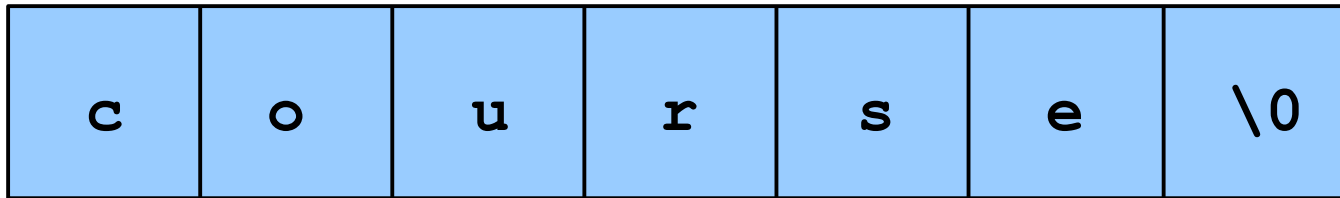
++pa ⇒ increments **pa** to point to next **int**

Example - String Comparison

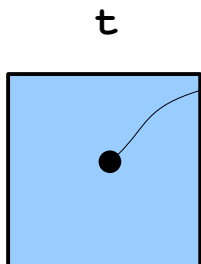
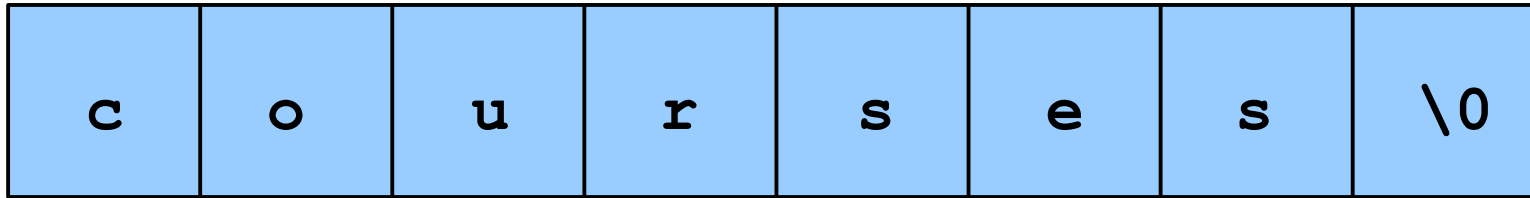
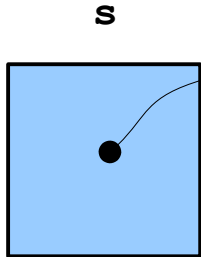
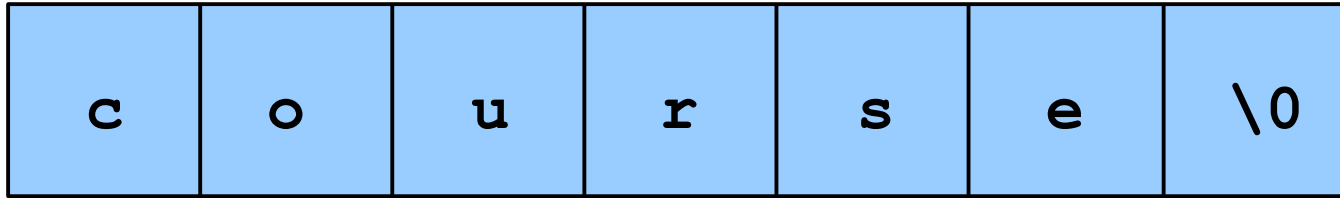
```
#define MAXLEN 100
int my_strcmp(char *p1, char *p2);
/*exactly the same as int my_strcmp(char p1[], char p2[]);*/
int getline(char *line, int max);
/*exactly the same as int my_getline(char line [], int max);*/

int main() {
    char a[MAXLEN], b[MAXLEN];
    int comp;
    my_getline(a, MAXLEN);
    my_getline(b, MAXLEN);
    comp = my_strcmp(a, b);
    if(comp > 0)
        printf("First line is greater than second\n");
    else if (comp < 0)
        printf("First line is less than second\n");
    else printf("These lines are equal!\n");
    return 0
}
```

Example - String Comparison



Example - String Comparison



Compare String Comparisons

```
/* pointer version */
/* my_strcmp: return <0 if s<t,
   0 if s==t, >0 if s>t */
int my_strcmp(char *s, char *t)
{
    for( ; *s == *t; s++, t++)
        if(*s == '\0')
            return 0;
    return *s - *t;
}
```

```
/* array version */
/* my_strcmp: return <0 if s<t,
   0 if s==t, >0 if s>t */
int my_strcmp(char *s, char *t)
{
    int i;

    for(i = 0 ; s[i] == t[i]; i++)
        if(s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

Pointer Subtraction

- Pointer subtraction is also valid

```
size_t strlen(const char* s)
{
    const char* p = s;
    while(*p != '\0')
        ++p;
    return p - s;
}
```

NULL Pointer

- 0 is a special value we can assign to a pointer which does not point to anything
 - most frequently, a symbolic constant **NULL** is used
- It is guaranteed, that no valid address is equal to 0
- The bit pattern of the NULL pointer does not have to contain all zeros
 - usually it does
 - depends on the processor architecture
- On many machines, dereferencing a NULL pointer causes a segmentation violation

Pointer Traps and Pitfalls

- NULL ptr is not the same as an EMPTY string

```
const char* psz1 = 0;  
const char* psz2 = "";  
assert(psz1 != psz2);
```

- Always check for NULL before dereferencing a pointer

```
if (psz1)  
    /* use psz1 */
```

- `sizeof(psz1)` doesn't give you the number of elements in `psz1`. Need additional size variable.

Dangling Reference Bug

- Be careful not to return a pointer to a local variable
- Local variables are automatic, and are no longer valid after the function returns

```
#include <ctype.h> /* toupper */

const char* upcase(const char* s)
{
    char szBuf[100];
    char* p = szBuf;

    while(*p++ = toupper(*s++))
        /* empty */ ;
    return szBuf;
}
```

Dangling Reference Bug - Poor Solution

- Static variable will be valid after the function returns
- Still problems with this solution

```
#include <ctype.h> /* toupper */
#include <stdio.h> /* printf */

const char* upcase(const char* s)
{
    static char szBuf[100];
    char* p = szBuf;

    while(*p++ = toupper(*s++))
        /* empty */ ;
    return szBuf;
}

int main()
{
    printf("%s %s\n", upcase("ala"), upcase("ola"));
    return 0;
}
```

Function Pointers

- A function itself is not a variable
- However, you can take the address of a function
 - Store it in a variable
 - Store it in an array (e.g. C++ vtable emulation)
 - Pass it as a parameter to another function (e.g. `bsearch`, `qsort`). Aka, "callbacks"
 - Return it from a function (e.g. `GetProcAddress()` for Windows)
- Makes code independent of the actual function name and implementation

Function Pointer Variables

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

int main()
{
    float radians = 0, result = 0;
    char op = '\0';

    double (*pfn)(double) = 0; /* pfn is a function ptr variable */

    printf("Enter radians, a space, then s, c, or t: ");
    scanf("%g %c", &radians, &op);

    if (op == 's')        pfn = &sin;
    else if (op == 'c')  pfn = &cos;
    else if (op == 't')  pfn = &tan;

    result = (*pfn)(radians); /* Call function that pfn points to*/
    printf("Result=%g\n", result);
    return 0;
}
```

Function Pointer typedefs

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

typedef double (*PFN_OPER) (double);

int main()
{
    float radians = 0, result = 0;
    char op = '\0';

    PFN_OPER pfn = 0; /* pfn is a function ptr variable */

    printf("Enter radians, a space, then s, c, or t: ");
    scanf("%g %c", &radians, &op);

    if (op == 's')        pfn = &sin;
    else if (op == 'c')  pfn = &cos;
    else if (op == 't')  pfn = &tan;

    result = (*pfn)(radians); /* Call function that pfn points to*/
    printf("Result=%g\n", result);
    return 0;
}
```

Complicated Pointer Declarations

- `char** argv`
 - pointer to pointer to char
- `int *x[15]`
 - array of 15 pointers to integers
- `int (*x)[15]`
 - pointer to an array of 15 integers

Complicated Declarations

- `int *f()`
 - function returning pointer to an `int`
- `int (*f)()`
 - pointer to function returning an `int`
- `void *f()`
 - function returning a pointer to `void` (pointer to unknown type)
- `void (*f)()`
 - pointer to a function returning `void`
- `cdecl` program can help here

Command Line Arguments

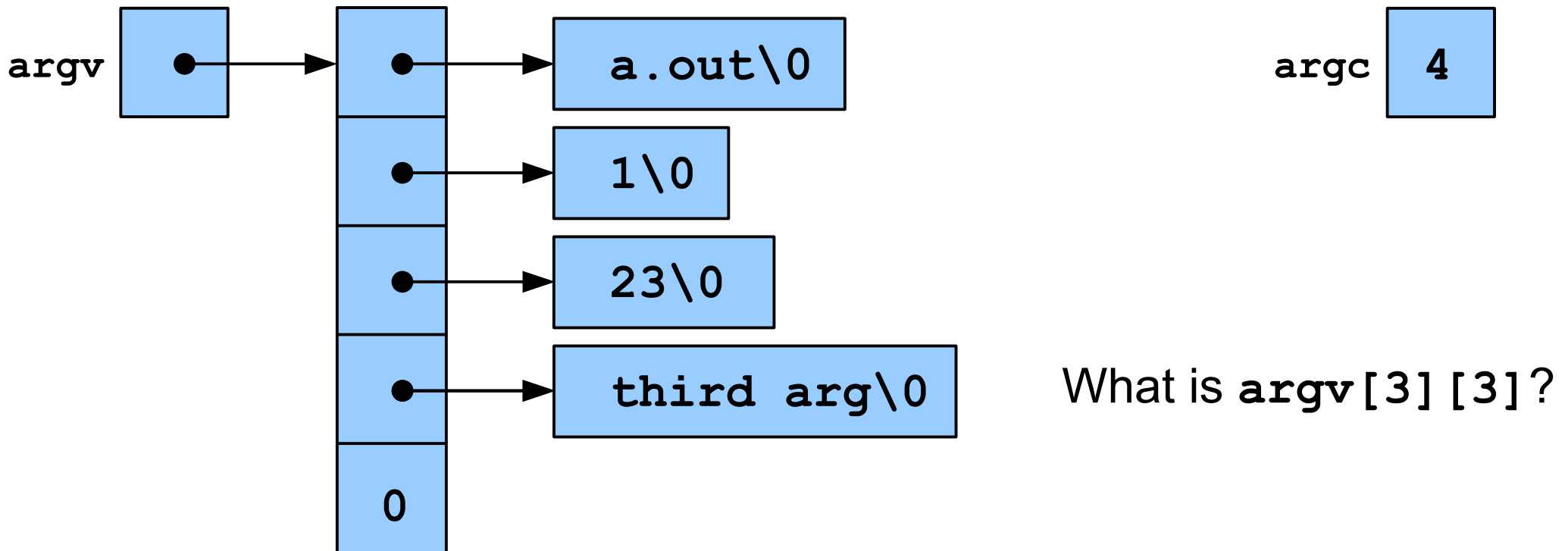
- Full declaration of `main` is as follows

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

- `argv` holds the command-line arguments, `argc` their number

```
$ a.out 1 23 "third arg"
```

- `argv` is a pointer to an array of strings



Command Line Arguments

- Print out the command-line arguments

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int i;
    for(i = 1; i < argc; ++i)
        printf("%d: %s\n", i, argv[i]);
    return 0;
}
```