MPHS
Introductory Elements

Carlo Caione
(carlo.caione@unibo.it)
Paolo Burgio
(paolo.burgio@unibo.it)

Why C

- This is NOT a C course.
- Why C:
  - Small size
  - Loose typing
  - External standard library – I/O, other facilities
  - Low level (BitWise) programming readily available
  - Pointer implementation - extensive use of pointers for memory, array, structures and functions
  - Macro preprocessor
  - It can handle low-level activities
  - It produces efficient programs
  - It can be compiled on a variety of computers

What C is used for

- Systems programming:
  - OSes, like Linux
  - microcontrollers: automobiles and airplanes
  - embedded processors: phones, portable electronics, etc.
  - DSP processors: digital audio and TV systems •
  - ...
- In MPHS:
  - Embedded systems
  - GPU (+ Extensions)
  - Smart phones (C derivatives)
  - Compilers and tools
  - Mark (finally)

Bitwise Operators

- The combination of pointers and bit-level operators makes C useful for many low level applications and can almost replace assembly code
- Writing software for embedded systems requires the manipulation of data at addresses, and this requires manipulating individual bits or groups of bits.
- ~ (Bitwise Negation)

a	b	~a	~b
0	1	1	0
0x0F0F	0x0000	0xF0F0	0xFFFF

- **&** (AND Operator)

a	b	a & b	
1	0	0	
1	1	1	
0xBEEF	0xFFF0	0xBEE0	Mask (clear) bits

Bitwise Operators

- **|** (OR Negation)

<table>
<tr><th>a</th><th>b</th><th>a | b</th><th></th></tr>
<tr><td>1</td><td>1</td><td>1</td><td></td></tr>
<tr><td>0</td><td>1</td><td>1</td><td></td></tr>
<tr><td>0x0EAD</td><td>0xD000</td><td>0xDEAD</td><td>Mask (set) bits</td></tr>
</table>

- **^** (XOR Operator)

a	b	a ^ b	
1	1	0	
0	1	1	
0xF0FA	0xFF00	0x0FFA	Toggling bits

Bitwise Operators

- Shifting Bits (**<<** left shift, **>>** right shift)

```
unsigned int i = 0xF0F0; // 1111 0000 1111 0000

unsigned int left = 0xF0F0 | (1 << 10); // 1111 0100 1111 0000
```

set the 11th bit

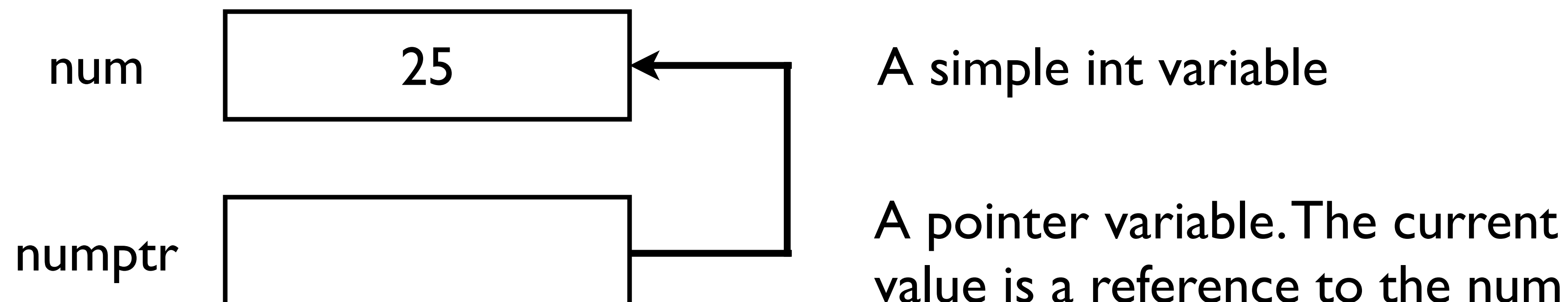
```
unsigned int i = 0x0F0F; // 0000 1111 0000 1111

unsigned int left = 0xF0F0 & (~(1 << 10)); // 0000 1011 0000 1111
```

clear the 11th bit

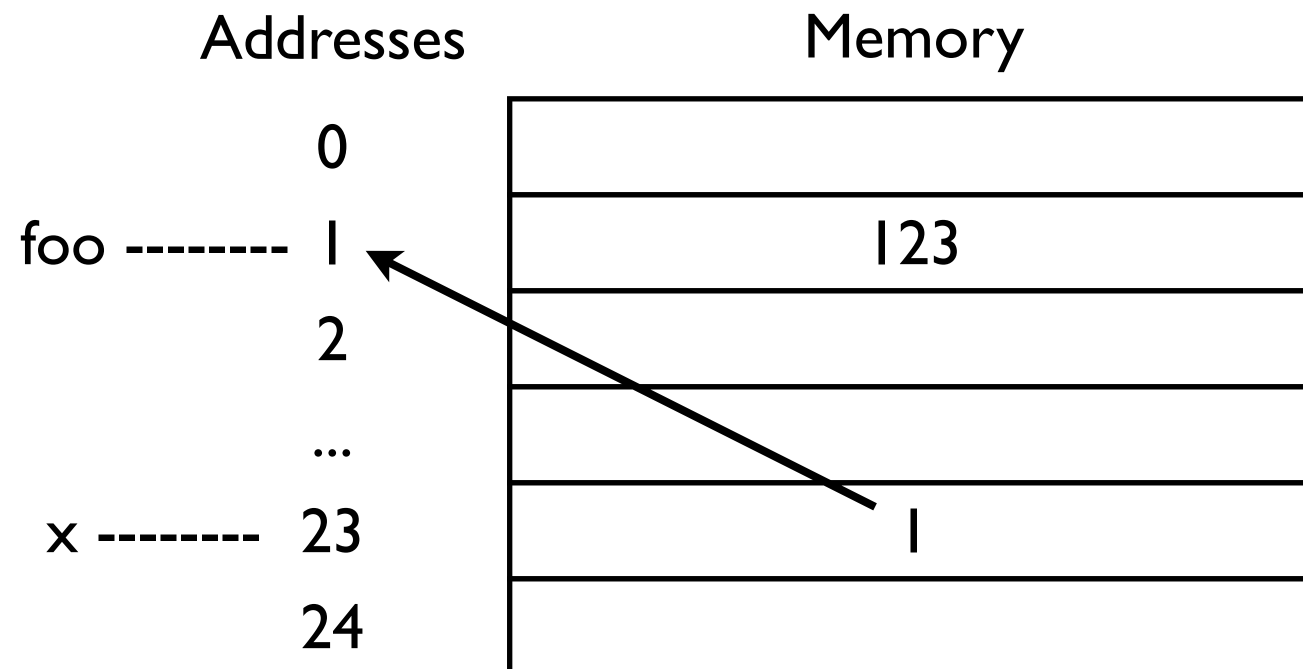
Pointers

- Why pointers?
  - allow different section of code to share information easily
  - enable complex data structures like linked list and binary trees
  - programs can manipulate addresses directly
- A pointed doesn’t store a simple value directly. Instead it store a reference, an address of something else
- &expr evaluates to the address of the location expr
- *expr evaluates to the value stored in the address expr

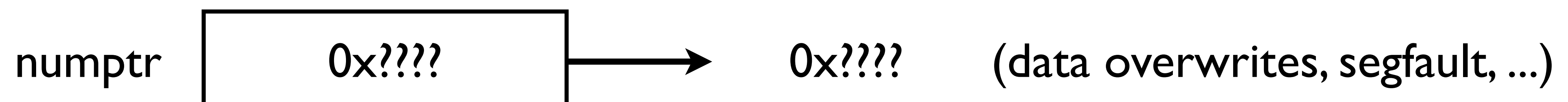


Pointers

```
int foo;
int *x;
foo = 123;
x = &foo;
```



- Pointer size depends on address space, not type
- When a pointer is first allocated it doesn't have a pointee. It points to a random memory address. Every pointer starts out with a bad value: each pointer must be assigned a pointee before it can support deference operations



Pointers

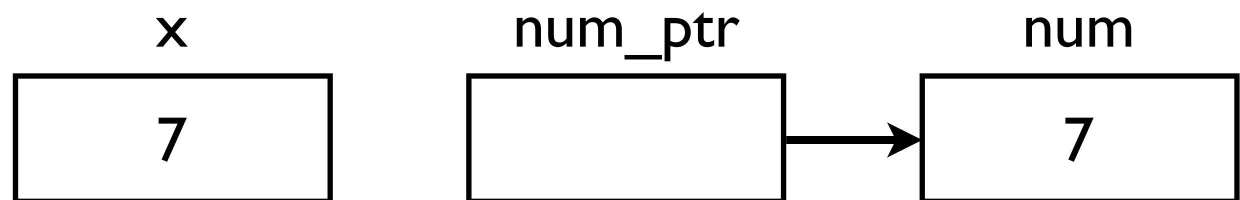
```
int num = 3;
int *num_ptr = NULL;
```

```
num_ptr = &num;
```

```
num = 5;
```

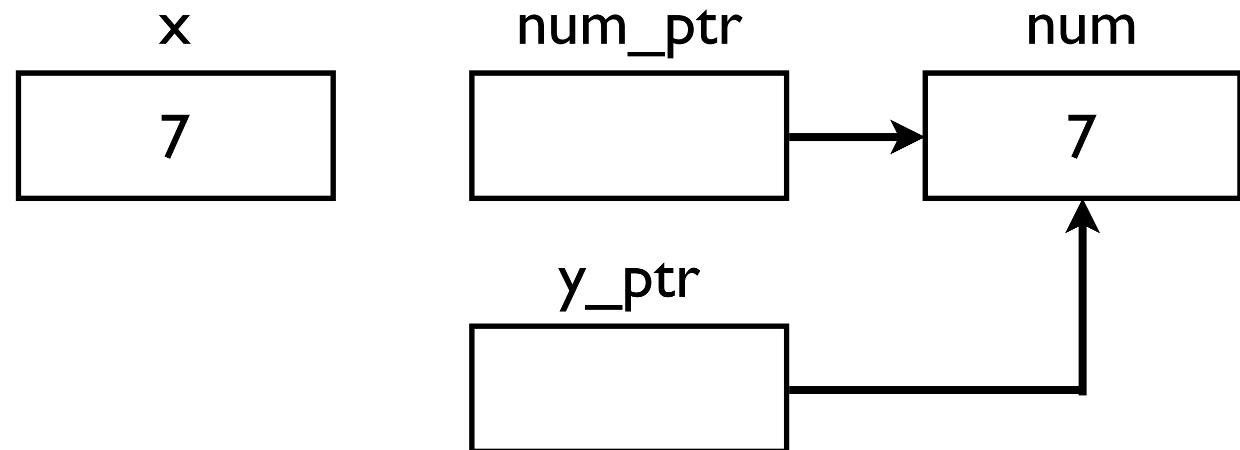
```
*num_ptr = 7;
```

```
int x = *num_ptr;
```

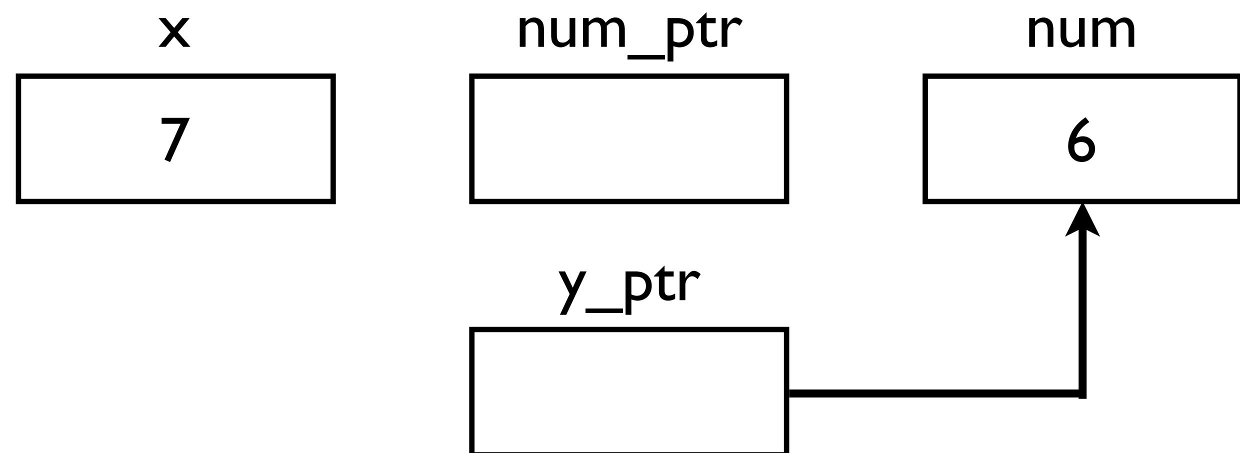


Pointers

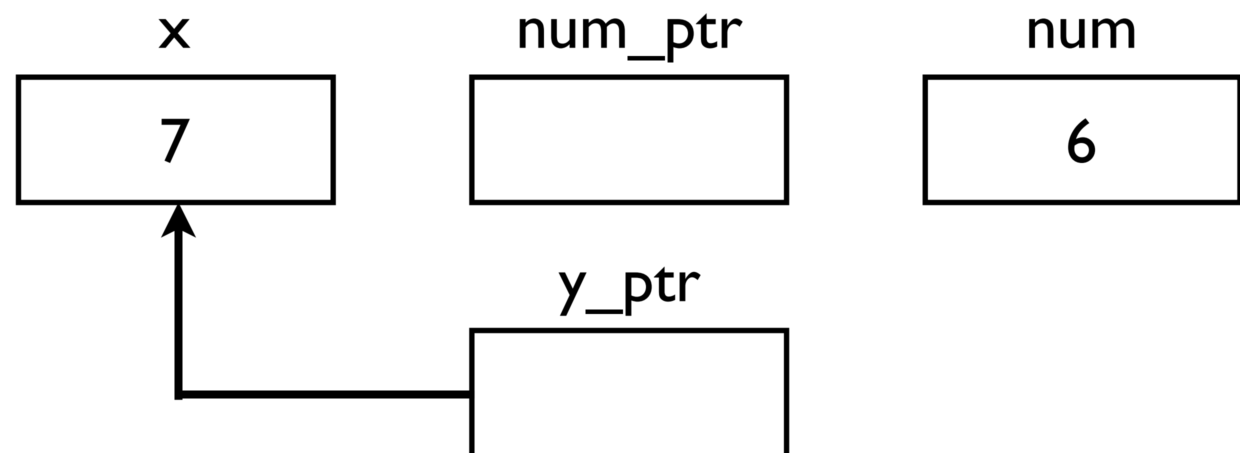
```
int *y_ptr = num_ptr;
```




```
*y_ptr = 6;
```

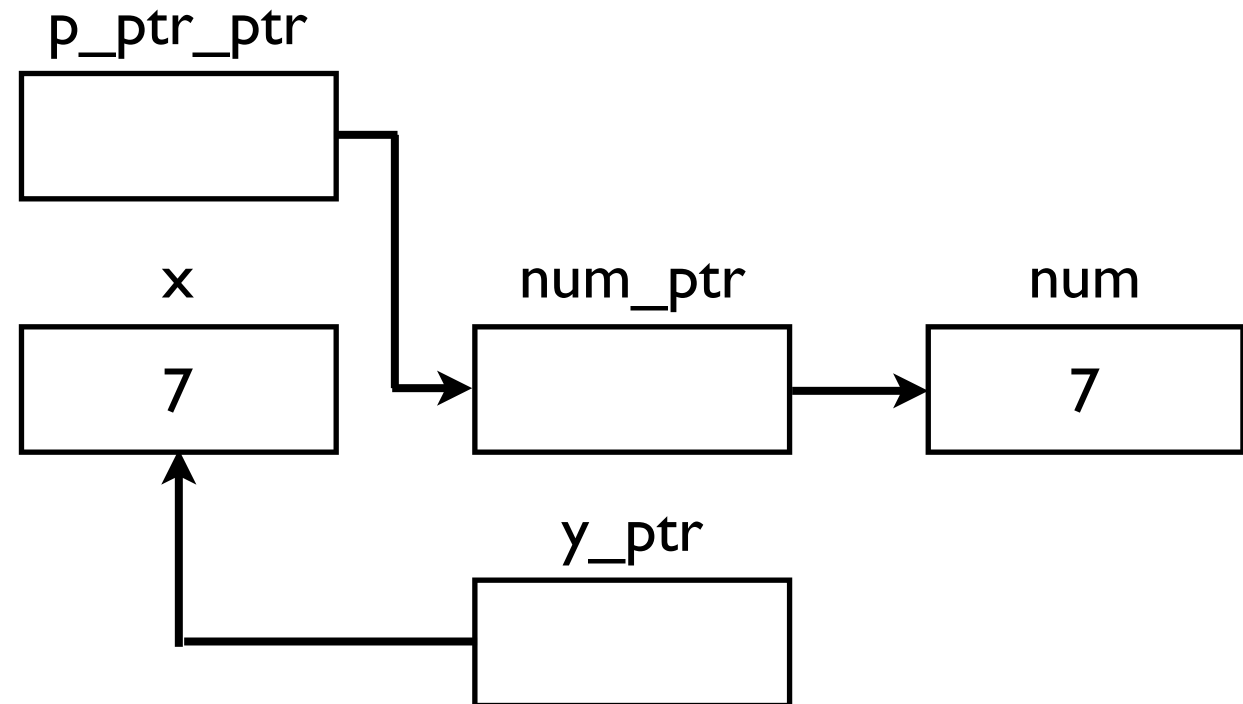



```
y_ptr = &x;
```

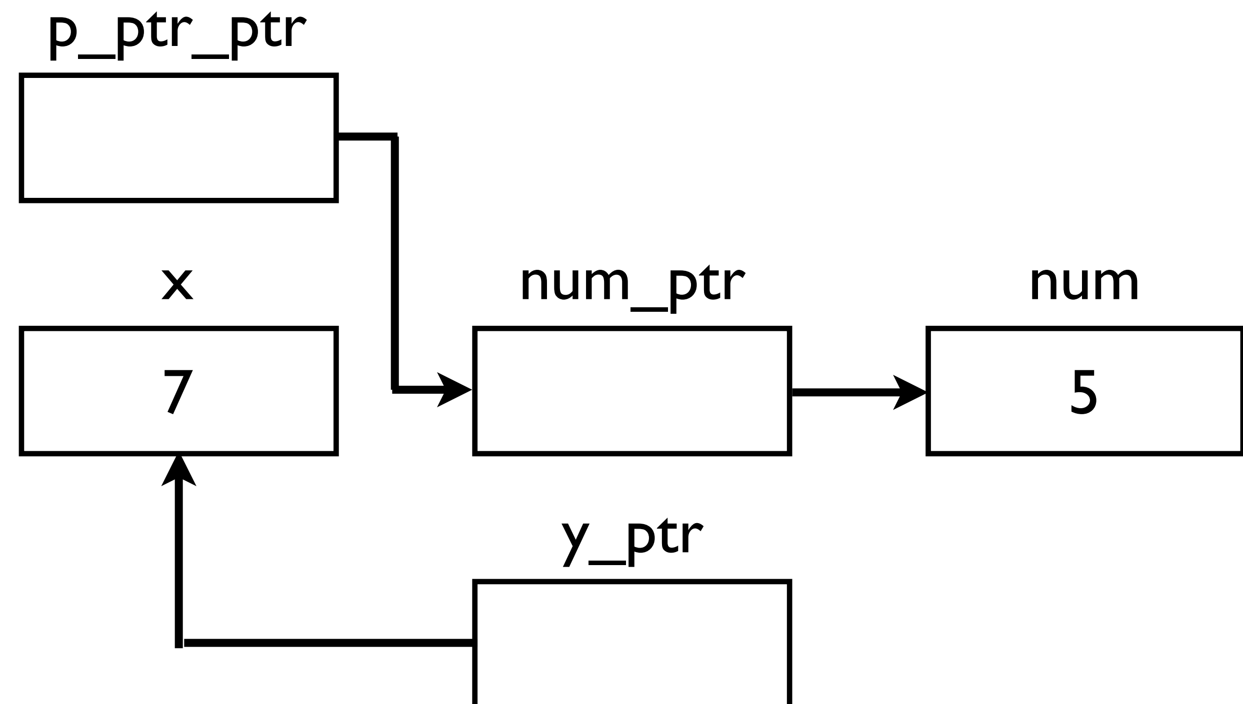


Pointers

```
int **p_ptr_ptr;
p_ptr_ptr = &num_ptr;
```



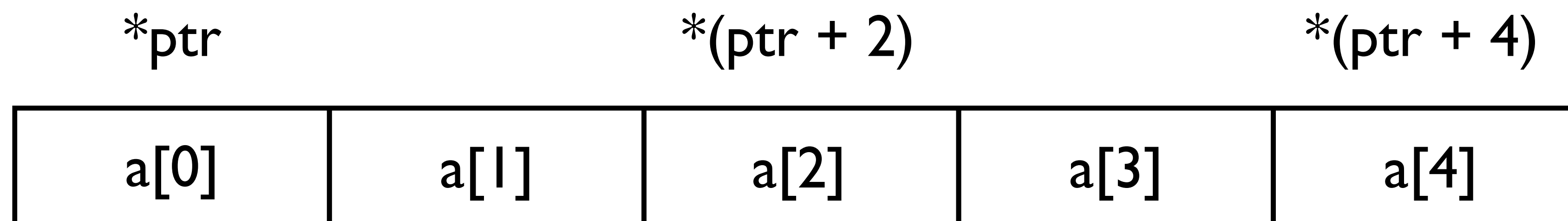

```
**p_ptr_ptr = 5;
```



Pointers

- Integer math operations can be used with pointers
- If you increment (or decrement) a pointer, it will be increased by the size of whatever it points to

```
int a[5];
int *ptr = a;
```



- Pointers are passed by value (the value of a pointer is the address it holds)

```
void swap(int *x, int *y){
   int tmp;
   tmp = *x;
   *x = *y;
   *y = tmp;
}
```

Pointers

- A useful technique is the ability to have pointers to functions

```
int (*func)(int a, float b);
```

- Once you've got the pointer, you can assign the address of the right sort of function just by using its name: a function name is turned into an address when it's used in an expression

```
void func(int);
main(){
   void (*fp)(int);
   fp = func;
   (*fp)(1);
}
void func(int){
   ...
}
```

- If you like writing finite state machines, you might like to know that you can have an array of pointers to functions

```
void (*fparr[])(int, float);
```

Organizing code files

- There are benefits to splitting up your project into several smaller files
- Firstly, look at how you would split your code into sections. Often this is by splitting it into separate subsystems, or 'modules', such as sound, music, graphics, file handling, etc
- This code to go in a header usually includes some or all of the following:
  - class and struct definitions
  - typedefs
  - function prototypes
  - global variables
  - constants
  - #define macros
  - #pragma directives

Organizing code files

- The key idea is that headers contain the interface, and the source files contain the actual implementation
- There are four basic errors that people encounter

1. The source files no longer compile as they can't find the functions or variables that they need.
2. Cyclic dependencies, where headers appear to need to #include each other to work as intended
3. Duplicate definitions where a class or identifier is included twice in a source file
4. Duplicate instances of objects within the code that compiled fine

Organizing code files (1)

- The first error, where a source file refuses to compile because one of the identifiers was undeclared, is easy to resolve
- Simply #include the file that contains the definition of the identifier you need

```
/* Header1.h */
#include "header2.h"
class ClassOne { ... };

/* Header2.h */
class ClassTwo { ... };

/* File1.cpp */
#include "Header1.h"
ClassOne myClassOne_instance;
ClassTwo myClassTwo_instance;
```

- The key here, is to explicitly #include any header files that you need for a given source file to compile

Organizing code files (2)

- Many constructs involve a two-way link of some sort, and this implies that both classes or structures know about each other

```
#include "child.h"
class Parent
{
    Child* theChild;
};

/* Child.h */
#include "parent.h"
class Child
{
    Parent* theParent;
};
```

- The Parent struct doesn't actually need to know the details of the Child class, as it only stores a pointer to one (Pointers are pretty much the same no matter what they point to)

Organizing code files (2)

- This is done with a forward declaration, taking the form of a class or class definition without a body

```
/* Parent.h */
class Child; /* Forward declaration of Child; */
class Parent
{
    Child* theChild;
};
```

- Notice how the #include line is replaced by the forward declaration
- As long as you are only referring to a pointer and not the actual type itself, you don't need to #include the full definition
- Otherwise: it is probably necessary to #include both parent.h and child.h in parent.c and child.c

Organizing code files (3)

- Duplicate definitions at compile time imply that a header ended up being included more than once for a given source file. This leads to a class or struct being defined twice, causing an error

```
/* Header1.h */
#include "header3.h"
class ClassOne { ... };

/* Header2.h */
#include "header3.h"
class ClassTwo { ... };

/* File1.cpp */
#include "Header1.h"
#include "Header2.h"
ClassOne myClassOne_instance;
ClassTwo myClassTwo_instance;
```

- For the purposes of compilation, File1.cpp ends up containing copies of Header1.h and Header2.h, both of which include their own copies of Header3.h

Organizing code files (3)

- This is where inclusion guards come in

```
#ifndef _INC_STDLIB
#define _INC_STDLIB
...
#endif  /* _INC_STDLIB */
```

- This ensures that the 'do something' only ever happens once
- Note that the symbol (in this case, "INC_FILENAME_H") needs to be unique across your project

Organizing code files (4)

- See next slide :-)

Storage class specifiers

- There are five keywords under the category of storage class specifiers: (`typedef`), `auto`, `extern`, `register` and `static`
- Only one storage class specifiers is permitted in a declaration. If you omit the storage class specifiers a default is chosen.
  - For external declarations the default storage class specifiers will be `extern`
  - For internal declarations it will be `auto`
  - For declaration of functions the default storage class is always `extern`
- Storage class specifiers are related to three different concepts:
  - Duration
  - Scope
  - Linkage

Storage class specifiers (duration)

- Duration of an object describes whether its storage is allocated once only, at the program start-up or is more transient in its nature
- There are only two types of duration of objects: static duration and automatic duration
  - Static (`static`) duration means that the object has its storage allocated permanently
  - Automatic (`auto`) duration means that the storage is allocated (freed) as necessary
- Data objects declared inside a functions are given the default storage class specifiers of `auto` unless some other storage class specifiers is used

```
void func(void){
   int a = 0;
   printf(“%d\n”,a++);
}

void main(void) {
   func();
   func();
}

output: 0 0
```

```
void func(void){
   static int a = 0;
   printf(“%d\n”,a++);
}

void main(void) {
   func();
   func();
}

output: 0 1
```

Storage class specifiers (duration)

- Register variables are special case of automatic variables. Automatic variables are allocated in the memory but accessing data in memory is considerably slow than processing in the CPU (registers)
- The `register` class specifier suggests to the compiler that it would be a good idea to store the object in one or more hardware registers in the interests of speed
- It is tending to fall into disuse nowadays
- It is illegal to get the address of a register variable

Storage class specifiers (scope)

- Scope defines when and where a given name has a particular meaning
- The different types of scope are the following:
  - function scope (labels)
  - file scope
  - block scope
  - function prototype scope
- Any name declared outside a function has file scope which means that the name is usable at any point from the declaration the the end of the source code
- A name declared inside a compound statement has block scope and is usable up to the end of the associated `}` which closes the compound statement

Storage class specifiers (scope)

```
int a = 0; // file scope
int b = 2;

int main(void){
   int c = 3; // block scope
   int b = 4;
   {
      int c = 5; // block scope
      printf("a = %d, b = %d, c = %d\n", a, b, c);
   }
}

output: a = 0, b = 4, c = 5
```

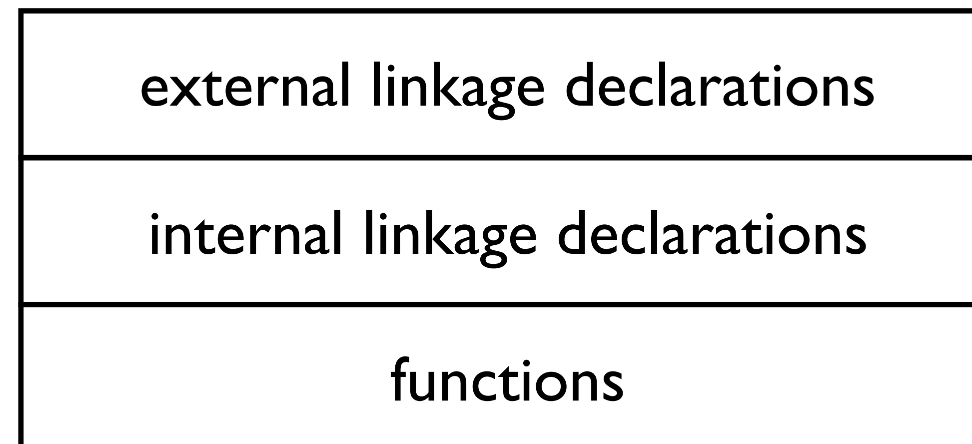
- Any declaration of a name within a compound statement hides any outer declaration of the same name until the end of the compound statement

Storage class specifiers (linkage)

- Linkage is used to determine what makes the same name declared in different scopes refer to the same thing
- The three different types of linkage are:
  - external linkage
  - internal linkage
  - no linkage
- In an entire program, built up from a number of source files and libraries, if a name has external linkage, then every instance of a that name refers to the same object throughout the program
- If something has internal linkage, it is only within a given source code file that instances of the same name will refer to the same thing
- Names with no linkage will refer to separate things

Storage class specifiers (linkage)

- The rules that determine the linkage and definition associated with declarations are quite complicated
- The recommended practice for the external and internal linkage is to declare all of the names in each relevant source files before you define any functions



- The external linkage declarations would be prefixed with `extern`
- The internal linkage declarations with `static`

Storage class specifiers (linkage)

```
#include <stdio.h>

// declarations with external linkage (definition somewhere else)
extern int important_var;
extern int library_func(double, int);

// definitions with external linkage
extern int ext_int_def = 0;

// internal linkage: only accessible inside this file
static int less_important_var;
static int local_func(int a1, int a2){
  return(a1 + a2);
}

static void lf(void){
  static int count; // static variable with no linkage
  int i = 1; // automatic variable with no linkage
}
```

Storage class specifiers (linkage)

- If the program is in several source files, and a variable is defined in file1 and used in file2 and file3, then extern declarations are needed in file2 and file3 to connect the occurrences of the variable.
- The usual practice is to collect extern declarations of variables and functions in a separate file, historically called a header
- file1:

```
int GlobalVariable;
void SomeFunction();
int main(){
   GlobalVariable = 1;
   SomeFunction();
   return(0);
}
```

- file2:

```
extern int GlobalVariable;
void SomeFunction(){
   ++GlobalVariable;
}
```

Type qualifiers (const)

- The keywords `const` and `volatile` can be applied to any declaration, including those of structures, unions, enumerated types or typedef names. Applying them to a declaration is called qualifying the declaration
- `const` means that something is not modifiable, so a data object that is declared with const as a part of its type specification must not be assigned to in any way during the run of a program
- For example, if you were accessing a hardware port at a fixed memory address and promised only to read from it, then it would be declared to be `const`
- Taking the address of a data object of a type which isn't `const` and putting it into a pointer to the `const`-qualified version of the same type is both safe and explicitly permitted

```
const int ci = 123;
const int *cpi;

cpi = &ci
```

- You will be able to use the pointer to inspect the object, but not modify it

Type qualifiers (const)

- Putting the address of a const type into a pointer to the unqualified type is much more dangerous (you can modify memory location)

```
const int ci = 123;
int *ncpi;
ncpi = &ci;
*ncpi = 0; // !!!! dangerous
```

- The main intention of introducing const objects was to allow them to be put into read-only store, and to permit compilers to do extra consistency checking in a program.

- But

```
char c;
char * const cp = &c;
```

- The const means that cp is not to be modified, although whatever it points to can be (the pointer is constant, not the thing that it points to)

Type qualifiers (volatile)

- After `const`, we have `volatile`. The reason for having this type qualifier is mainly to do with the problems that are encountered in real-time or embedded systems programming using C

- In general, any data that may be undated asynchronously should be declared to be volatile

- The reason to use `volatile` is to ensure that the compiler generates code to re-load a data item each time it is referenced in your program. Without `volatile`, the compiler may generate code that merely re-uses the value it already loaded into a register

- It advises the compiler that the data may be modified in a manner that may not be determinable by the compiler. This could be, for example, when a pointer is mapped to a device's hardware registers. The device may independently change the values unbeknownst to the compiler.

```
while((mem_loc & (READY | ERROR)) == 0)
   ;
```

Libraries

- In computer science , a library is a collection of subroutines or classes used to develop software
- Libraries contain code and data that provide services to independent programs. This allows the sharing and changing of code and data in a modular fashion
- Executables and libraries make references known as links to each other through the process known as linking, which is typically done by a linker
- Two types of libraries:
  - Static libraries
  - Dynamic / shared libraries

Libraries (static)

- Static libraries are simply a collection of ordinary object files; conventionally, static libraries end with the ``.a'' suffix
- Static libraries aren't used as often as they once were, because of the advantages of shared libraries
- But in embedded
- Static libraries permit users to link to programs without having to recompile its code, saving recompilation time
- Static libraries are often useful for developers if they wish to permit programmers to link to their library, but don't want to give the library source code
- For example, the C math library is typically stored in the file '/usr/lib/libm.a' on Unix-like systems. The corresponding prototype declarations for the functions in this library are given in the header file '/usr/include/math.h' (lib + header)

Libraries (static)

- Here is an example program which makes a call to the external function sqrt in the math library ‘libm.a’

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

int
main (void)
{
  double x = sqrt (2.0);
  printf ("The square root of 2.0 is %f\n", x);
  return 0;
}
```

- To enable the compiler to link the sqrt function to the main program ‘calc.c’ we need to supply the library ‘libm.a’

```
gcc -Wall calc.c /usr/lib/libm.a -o calc

gcc -Wall calc.c -lm -o calc
```

Libraries (dynamic)

- Shared libraries are libraries that are loaded by programs when they start
- It's actually much more flexible and sophisticated than this, because the approach used by Linux permits you to:
    - update libraries and still support programs that want to use older, non-backward - compatible versions of those libraries
    - override specific libraries or even specific functions in a library when executing a particular program
    - do all this while programs are running using existing libraries
- If the linker finds that the definition for a particular symbol is in a shared library, then it doesn't include the definition of that symbol in the final executable
- Instead, the linker records the name of symbol and which library it is supposed to come from in the executable file instead

Libraries (dynamic)

- When the program is run, the operating system arranges that these remaining bits of linking are done "just in time" for the program to run. Before the main function is run, a smaller version of the linker (often called ld.so) goes through these promissory notes and does the last stage of the link there and then—pulling in the code of the library and joining up all of the dots

- If a particular symbol is pulled in from a particular shared library (say printf in libc.so), then the whole of that shared library is mapped into the address space of the program. This is very different from the behavior of a static library, where only the particular objects that held undefined symbols got pulled in

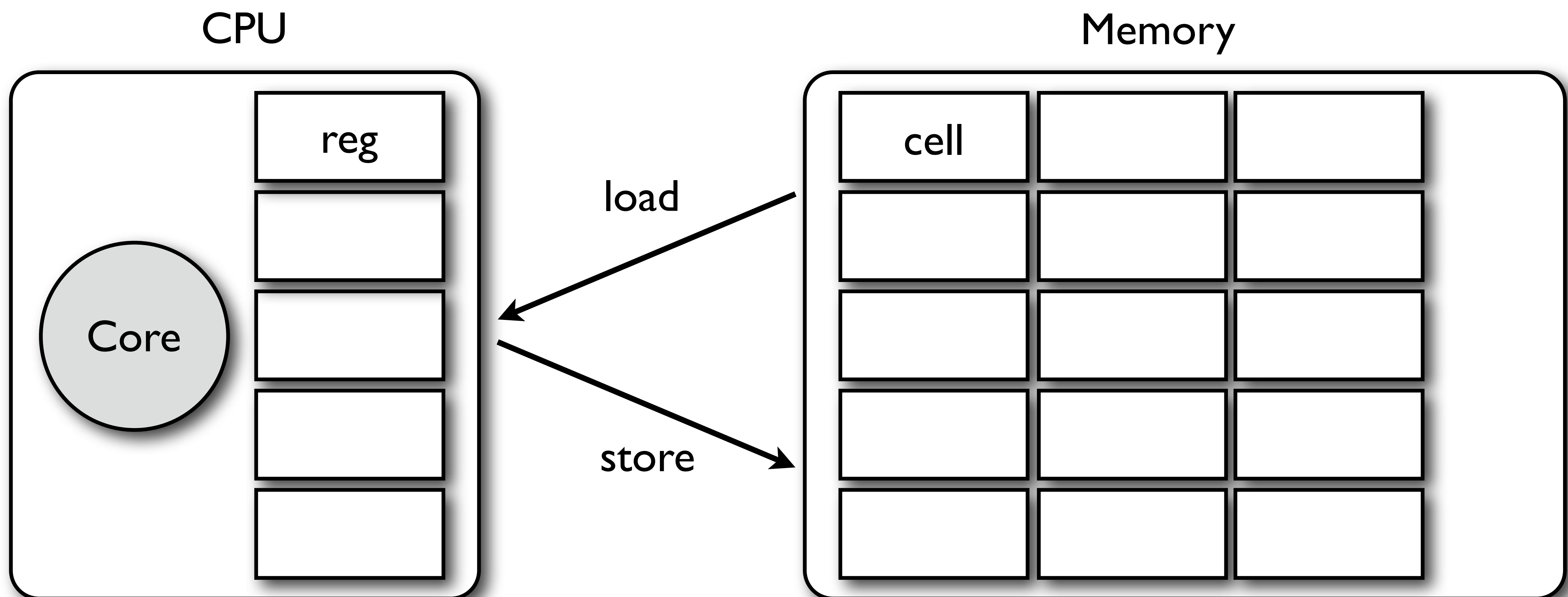
- Compiling for runtime linking with a dynamically linked libctest.so.1.0:

```
gcc -Wall -L/opt/lib prog.c -lctest -o prog
```

- The libraries will NOT be included in the executable but will be dynamically linked during runtime execution

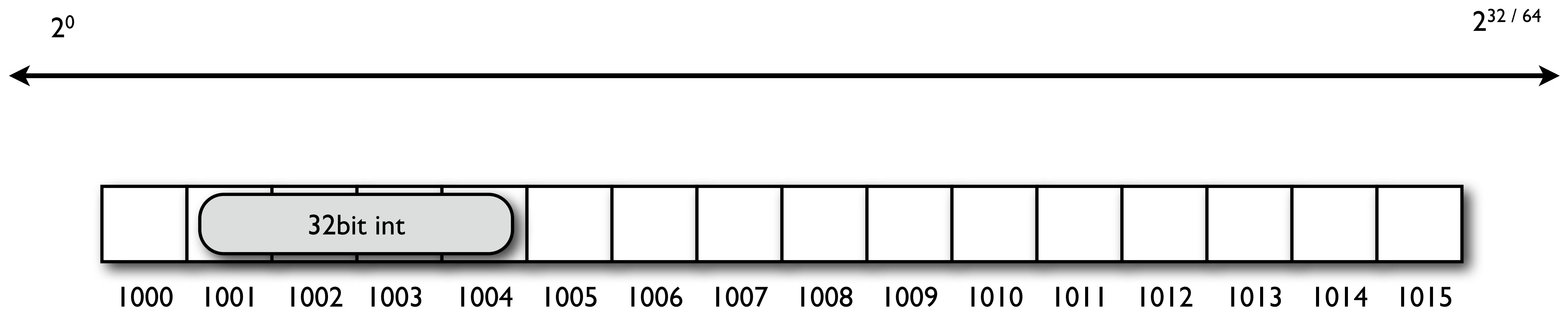
Memory

- CPU operates on data in a handful of registers
- These are not enough for most applications
- Memory is just a vast collection of value holders
- Values are copied to and from registers



Memory

- Programmers see linear address space
- Typical address space is vast 32 or 64 bits
- Size of cell is limited (typically 1 byte, 8 bits)
- Spread values in contiguous cells
- We need to know starting address and size

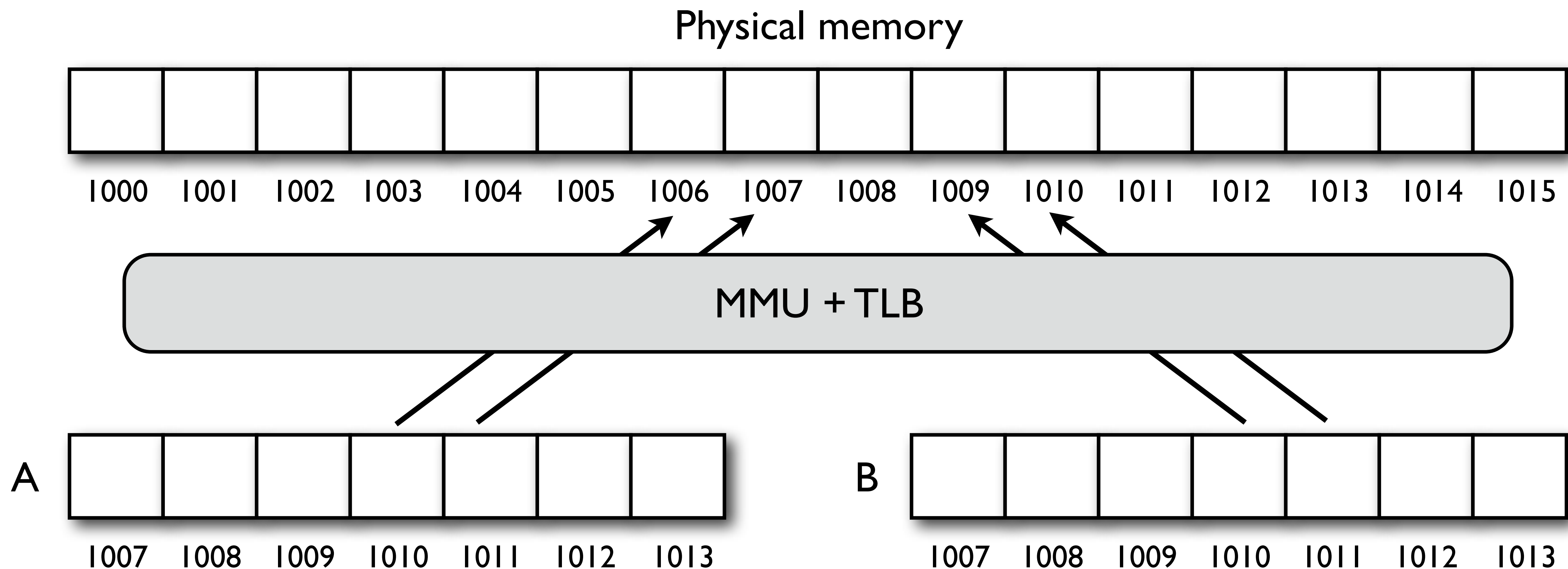


Little endian and big endian

- How do we split (and reassembly) multibyte values
- In computing endianness is the order of individually addressable sub-units within a longer data word
- Endianness may be seen as a low-level attribute of a particular representation format, for example, the order in which the two bytes are stored in memory
- Most modern computer processors agree on bit ordering "inside" individual bytes. This means that any single-byte value will be read the same on almost any computer one may send it to.
- Integers are usually stored as sequences of bytes, so that the encoded value can be obtained by simple concatenation. The two most common of them are:

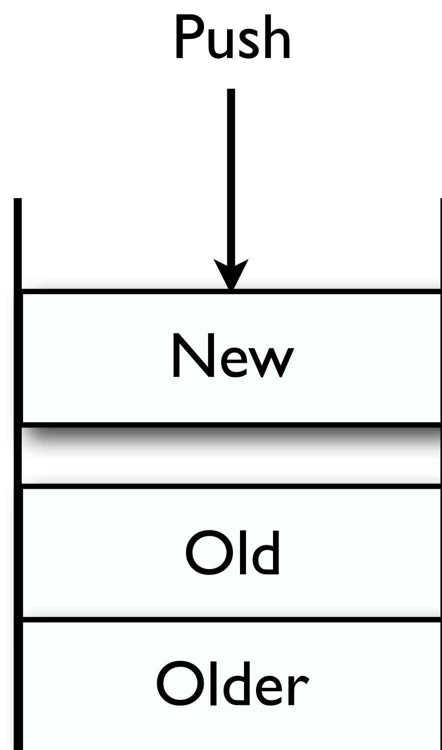
Virtual memory

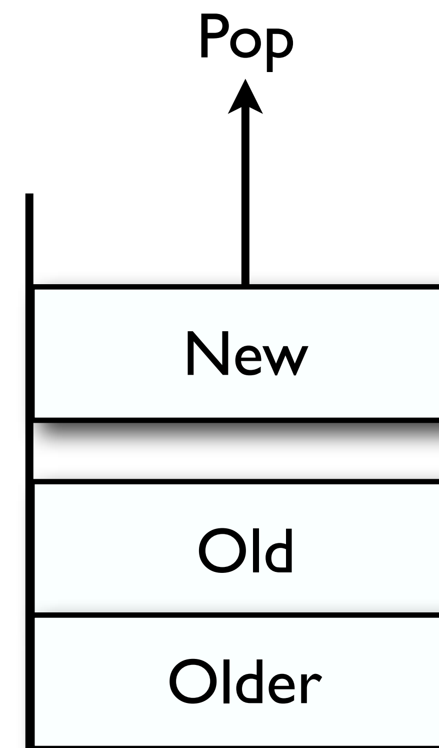
- Each process thinks it has a large (2^{32} - 2^{64}) range of contiguous addresses
- Certain logical addresses are mapped to physical addresses (RAM + disk file for inactive parts)
- The conversion mechanism is transparent
- Overlapping addresses do not interfere



Stack

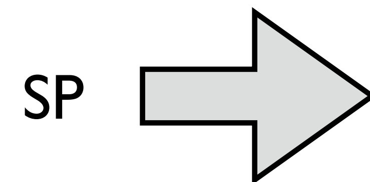
- Variables are allocated space on stack
- Stack is a LIFO (Last-In-First-Out)
- It has a fixed base and grows “up” or “down”
- Newer variables allocated “above” earlier
- Newer variables “die” before older

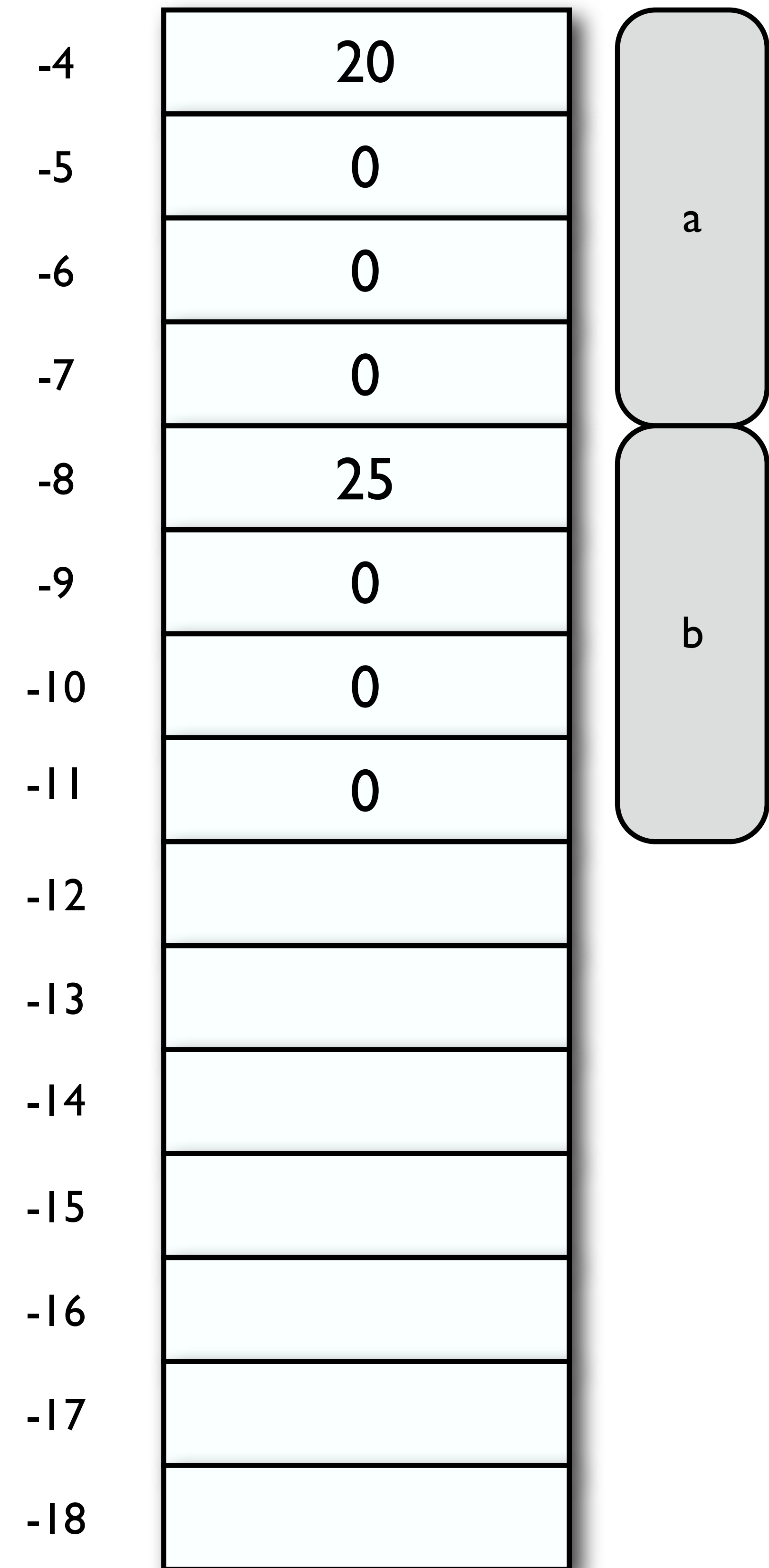




Stack

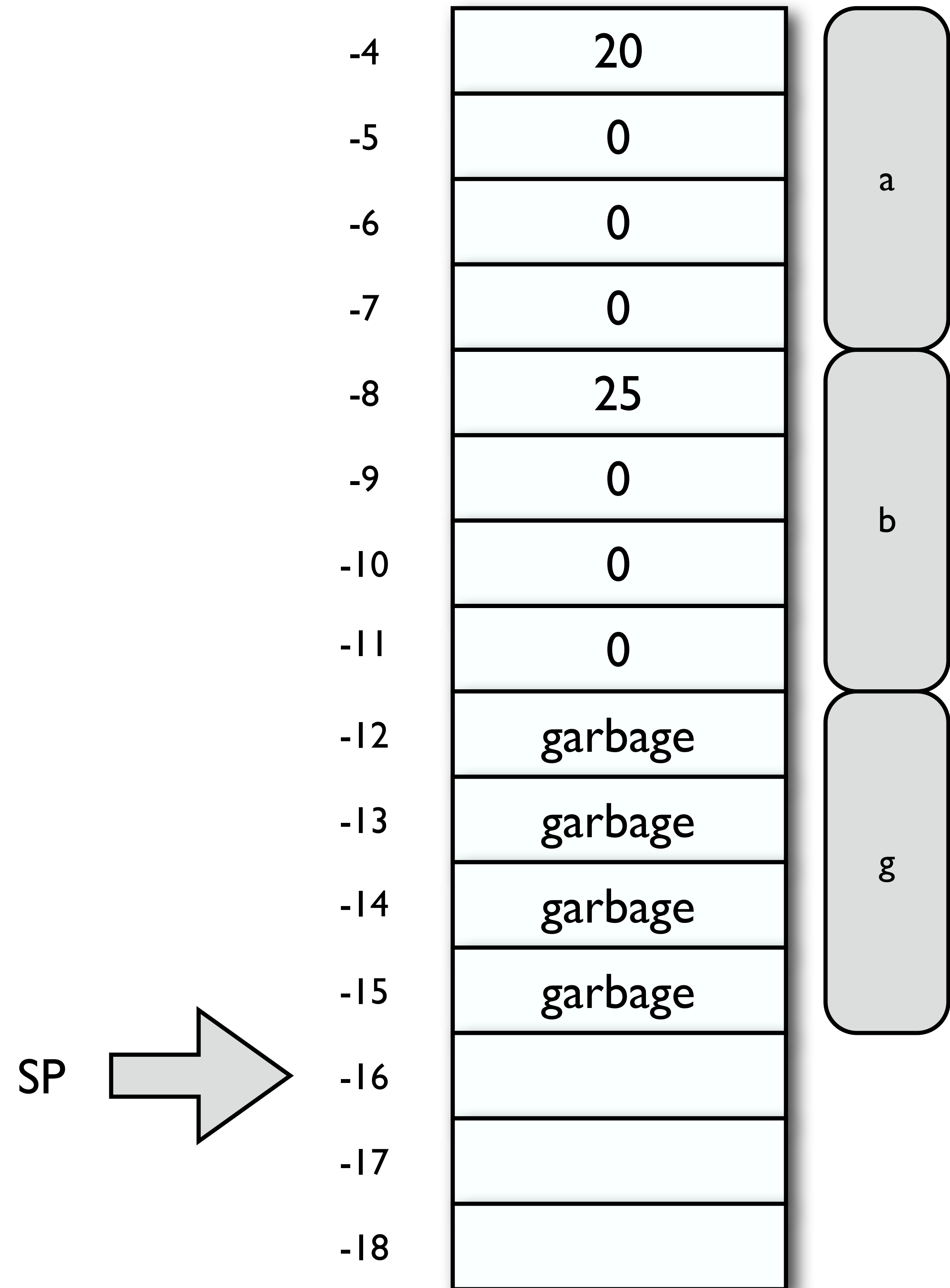
```
int main(void){
  int a = 20, b = 25;
  ...
```





Stack

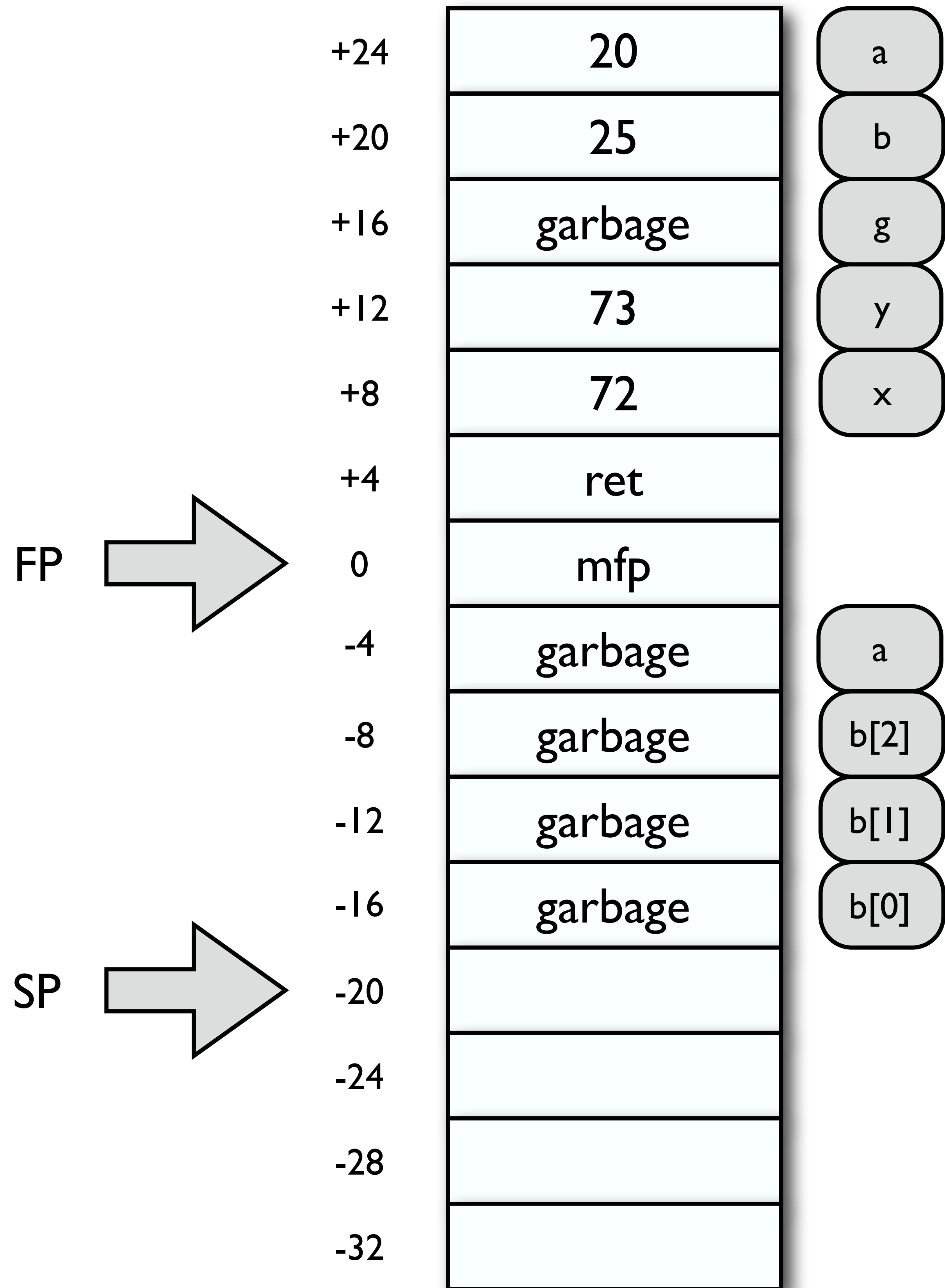
```
int main(void){
  int a = 20, b = 25;
  {
    int g;
    ...
```



Stack

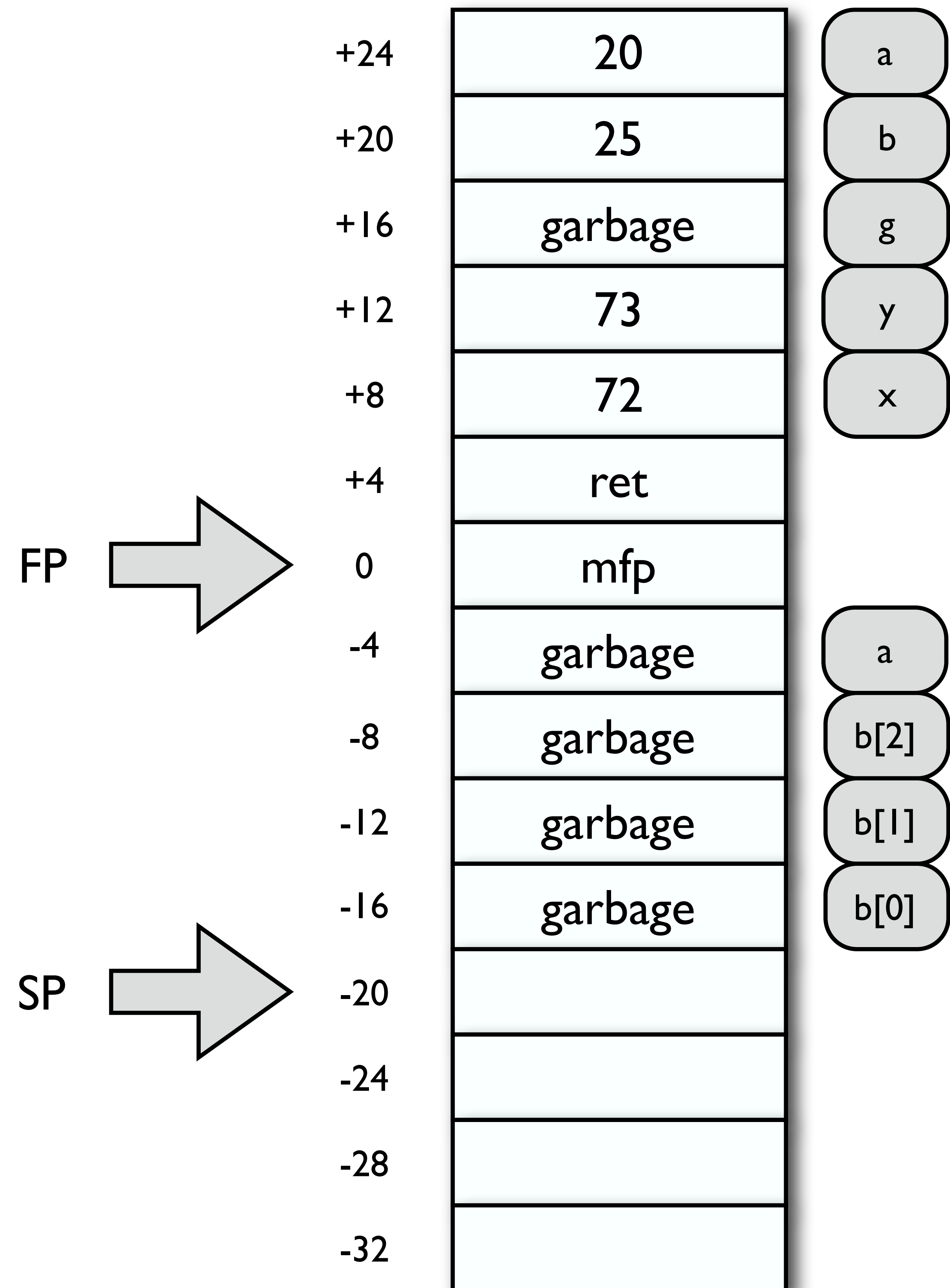
```
int main(void){
  int a = 20, b = 25;
  {
    int g;
    g = func(72,73);
  }
}

int func(int x, int y){
  int a;
  int b[3];
  ...
```



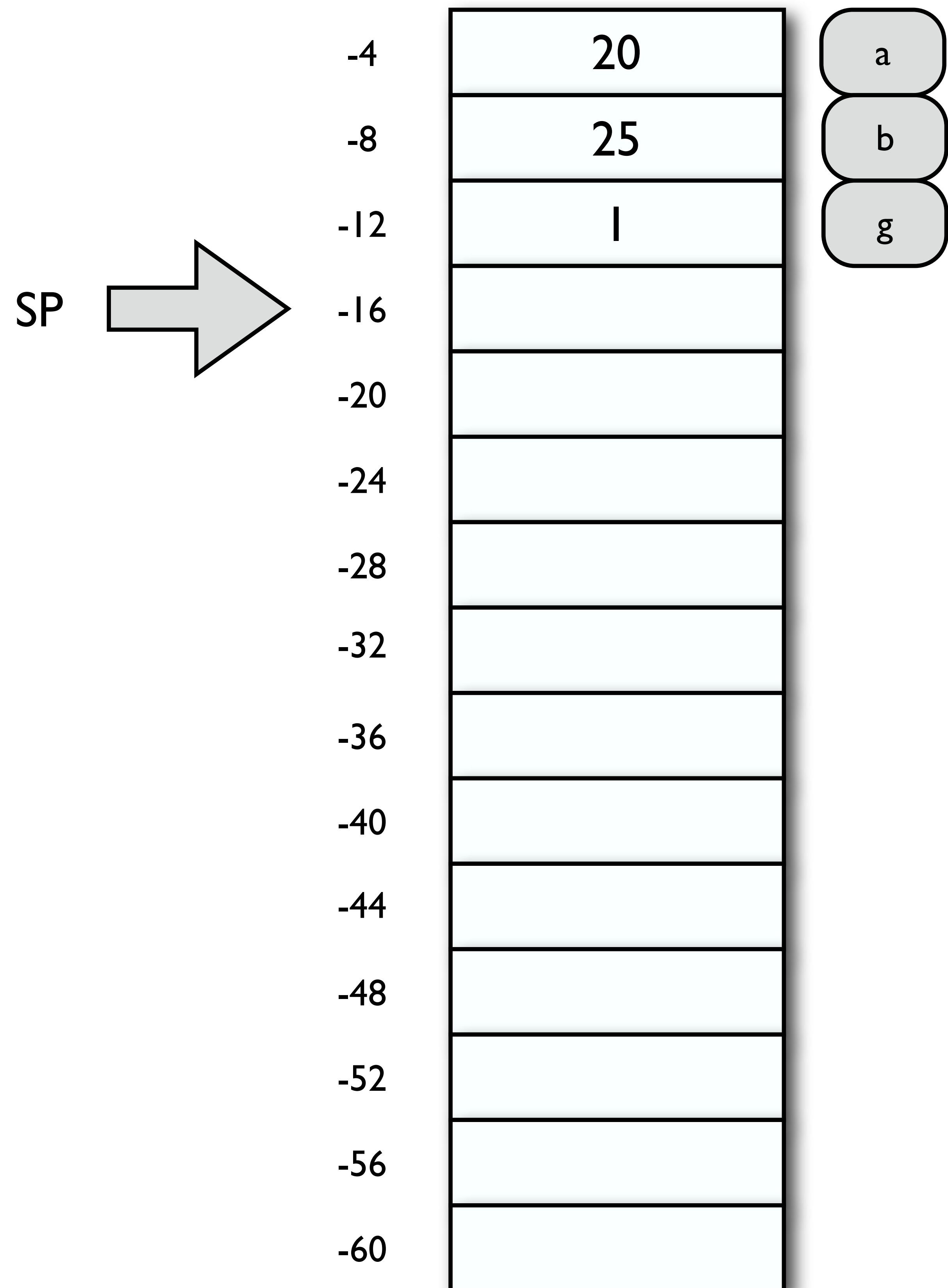
Stack

- Copies of argument values pushed into stack
- Return address pushed into stack
- CPU IP aimed at beginning of function block
- All auto variables and parameters are referenced via offsets from the frame pointer (FP)
- The FP and Stack Pointer (SP) are in registers



Stack

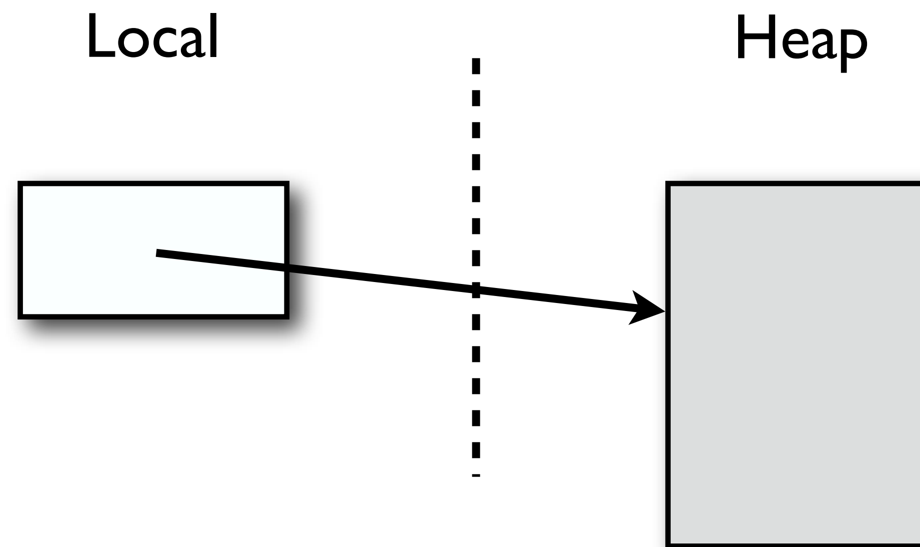
- When func returns, the return value is stored in a register
- The stack pointer is moved to the y location
- The code is jumped to the return address ret
- Frame pointer is set to mfp
- The caller moves the return value to the right place



Heap

- Heap memory, also known as "dynamic memory" is an alternative to local stack memory
- Why? Size of data may not be known in advance
  - May depend on user input
  - May depend on result of calculation
  - Size may change over time
- We want to control lifetime of data stored over time
- The heap is a large area of memory available for use by the program. The program can request areas or "blocks" of memory for its use within the heap.
- In order to allocate a block of some size the program makes an explicit request by calling the heap allocation function
- The allocation function reserves a block of memory of requested size in the heap and returns a pointer to it

Heap



- Each allocation request reserves a contiguous area of the requested size in the heap and return a pointer to that new block to the program
- The heap manager can allocate the blocks wherever it wants in the heap so long as the blocks do not overlap
- When the program is finished using a block of memory, it makes an explicit deallocation request. The block is free again and so may be reused to satisfy future allocations

Heap

- The library functions which make heap requests are `malloc()` and `free()` (prototypes in <stdlib.h>)

```
int *ar = NULL;
ar = (int *)malloc(30 * sizeof(int));
...
free(ar);

struct fraction *fracts;
fracts = malloc(10 * sizeof(struct fraction));
...
free(fracts);
```

- A program which forgets to deallocate a block is said to have a "memory leak"
- The result will be that the heap gradually fill up as there continue to be allocation requests
- Programmer is responsible for releasing dynamically allocated memory ASAP

Memory segmentation

- In a computer system using segmentation, an instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment. A segment has a set of permissions, and a length, associated with it
- When a program is executed it is read into memory where it resides until termination. The code allocates a number of special purpose memory blocks for different data types

STACK a very dynamic kind of memory located at it's top (high addresses) and growing downwards
memory not allocated yet Memory that will soon become allocated by the stack, that grows down. Stack will grow until it hits the administrative limit (predefined).
shared libraries
memory not allocated yet Memory that will soon become allocated by the heap growing up from underneath.
HEAP It is said that this is the most dynamic part of memory. It is dynamically allocated and freed in big chunks. The allocation process is rather complex (stub/buddy system) and is more time consuming than putting things on stack.
BSS Memory containing global variables of known (predeclared) size.
Constant data All constants used in a program.
Static program code
Reserved / other stuff

Memory segmentation

- In the PC architecture there are four basic read-write memory regions in a program:
  - **DATA.** The Data area contains global and static variables used by the program that are initialized
  - **BSS.** In C, statically-allocated variables without an explicit initializer are initialized to zero (for arithmetic types) or a null pointer (for pointer types). Implementations of C typically represent zero values and null pointer values using a bit pattern consisting solely of zero-valued bits (though this is not required by the C standard). Hence, the bss section typically includes all uninitialized variables declared at the file level (i.e., outside of any function) as well as uninitialized local variables declared with the static keyword
  - **HEAP.** The Heap area is managed by malloc, realloc, and free. The Heap area is shared by all shared libraries and dynamically loaded modules in a process.
  - **STACK.** The stack is a LIFO structure, typically located in the higher parts of memory. It usually "grows down" with every register, immediate value or stack frame being added to it

Memory segmentation

- And finally:
  - **CODE.** In computing, a code segment, also known as a text segment or simply as text, is a phrase used to refer to a portion of memory or of an object file that contains executable instructions. It has a fixed size and is usually read-only

Process (vs Thread)

- In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently

- In general, a computer system process consists of (or is said to 'own') the following resources:

  - An image of the executable machine code associated with a program

  - Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time

  - Operating system descriptors

  - Security attributes

  - Processor state (context), such as the content of registers, physical memory addressing, etc

Process (vs Thread)

- Under Unix and Unix-like operating systems, the parent and the child processes can tell each other apart by examining the return value of the fork() system call
- When a fork() system call is issued, a copy of all the pages corresponding to the parent process is created, loaded into a separate memory location by the OS for the child process
- But this is not needed in certain cases. Consider the case when a child executes an "exec" system call (which is used to execute any executable file from within a C program) or exits very soon after the fork()
- In some cases, a technique called copy-on-write (COW) is used. With this technique, when a fork occurs, the parent process's pages are not copied for the child process. Instead, the pages are shared between the child and the parent process. Whenever a process (parent or child) modifies a page, a separate copy of that particular page alone is made for that process (parent or child) which performed the modification

(Process vs) Thread

- Threads differ from traditional multitasking operating system processes in that:
  - processes are typically independent, while threads exist as subsets of a process
  - processes carry considerable state information, whereas multiple threads within a process share state as well as memory and other resources
  - processes have separate address spaces, whereas threads share their address space
  - processes interact only through system-provided inter-process communication mechanisms
  - Context switching between threads in the same process is typically faster than context switching between processes

(Process vs) Thread

Elements per process	Elements per thread
Address space	Program counter
Global variables	Registers
Open files	Stack
Children processes	State
Pending alarms	
Signals and signal handlers	
Account informations