Høyskolen Kristiania

PG3401
C Programmering for Linux

Lecture 4 (week 05)

Pointers

Course literature

- K&R, chapter 5; Pointers and arrays

Recap

- C:
  - C programming paradigm == procedure oriented
  - Basic introduction to coding in C
  - Commonly used elements in C
  - Warnings and possible errors
- Variable, operators and expressions
- Primitive data types
- Control structures

Memory

- Physical Memory : The actual physical resource that resides as hardware.
  - Cache
  - RAM
  - Disk
  - Other media
- Virtual Memory : Accessed by processes – abstraction by OS – simple addressable

Size of Physical Memory

- Cache – 4MB [intel i7-4578U]
- Memory (RAM) – 8GB [usual]
- Hard disk - 2 TB ++
- Many hardware controllers have cache as well

Physical Memory

- Different access speeds (RAM, cache)
- Usually, max size is not a constraint except :
  - Memory intensive programs
  - Embedded systems (no OS)

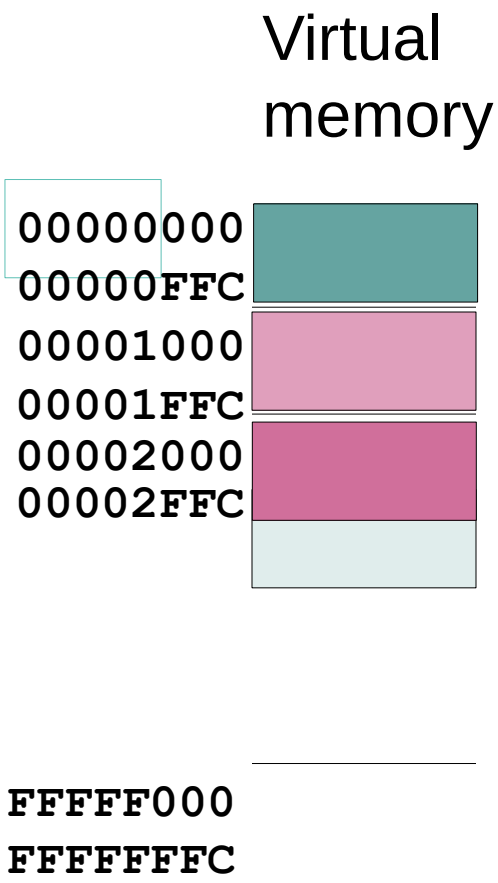
Size of Virtual Memory

- For 32-bit, 2^{32}, typically 2 – 3 GB of usable memory
- For 64-bit, 2^{64}, about a lot

- Hexadecimal representation:
  - 4 bits – one digit
  - 2 digits per byte – 0x00, 0xA0, 0x9F, 0x7E, 0xFF
  - 32 –bit - 4 bytes – 8 digits – 0xFFA2983E
  - 64 – bit - 8 bytes – 16 digits – 0xFF5EB2877E24A690

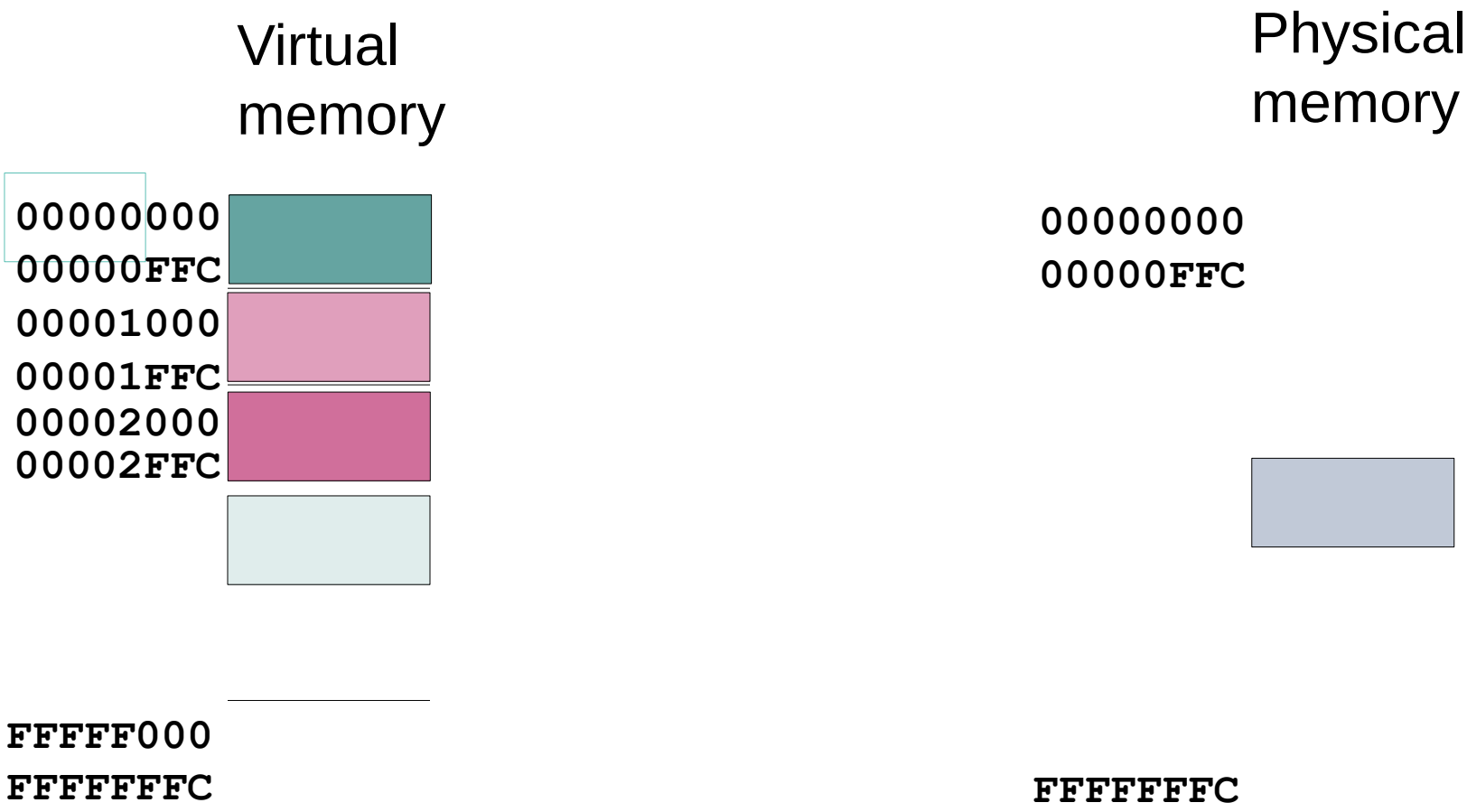
Paging – virtual to physical memory #1

Høyskolen Kristiania



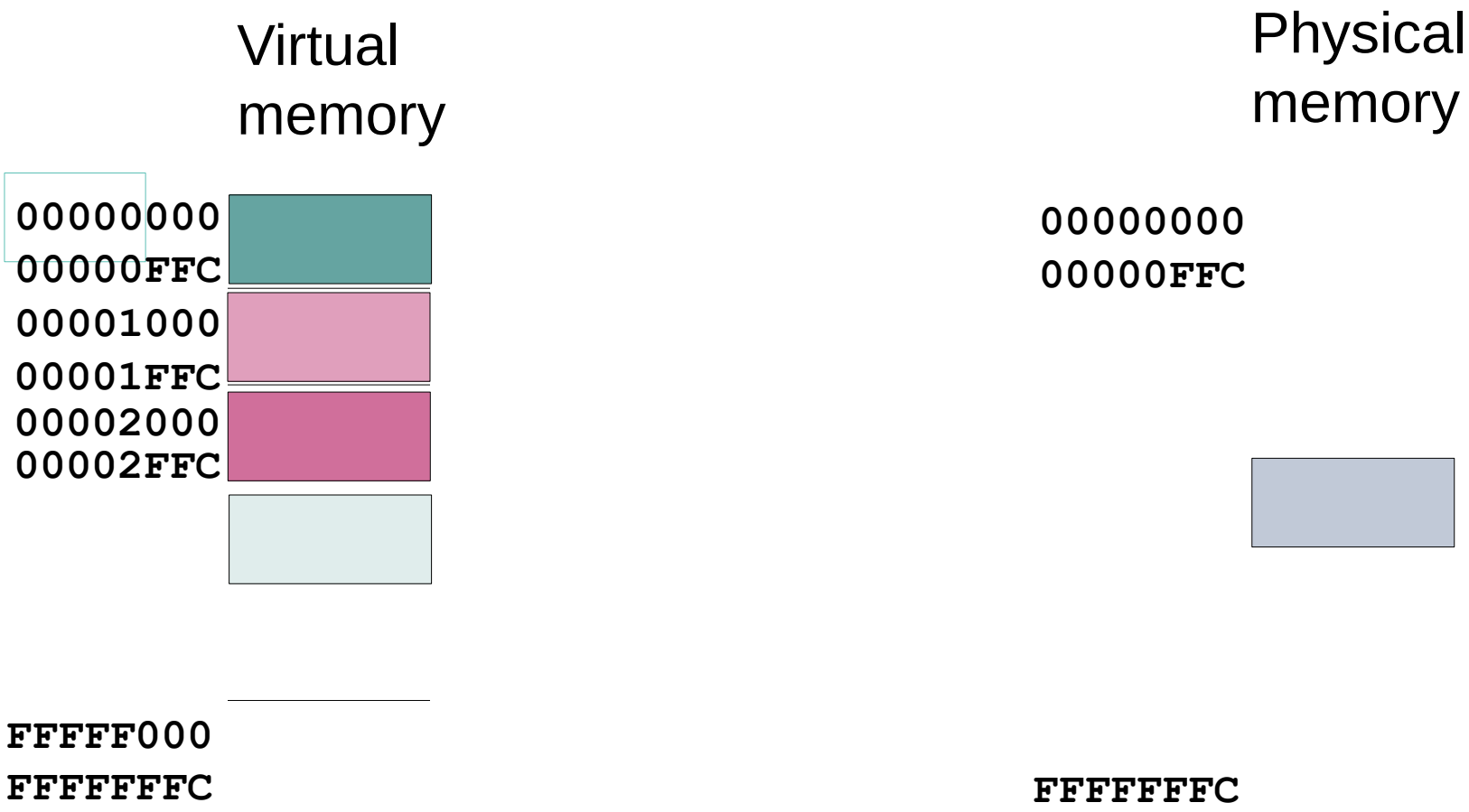
Paging – virtual to physical memory #2

Høyskolen Kristiania



Paging – virtual to physical memory #3

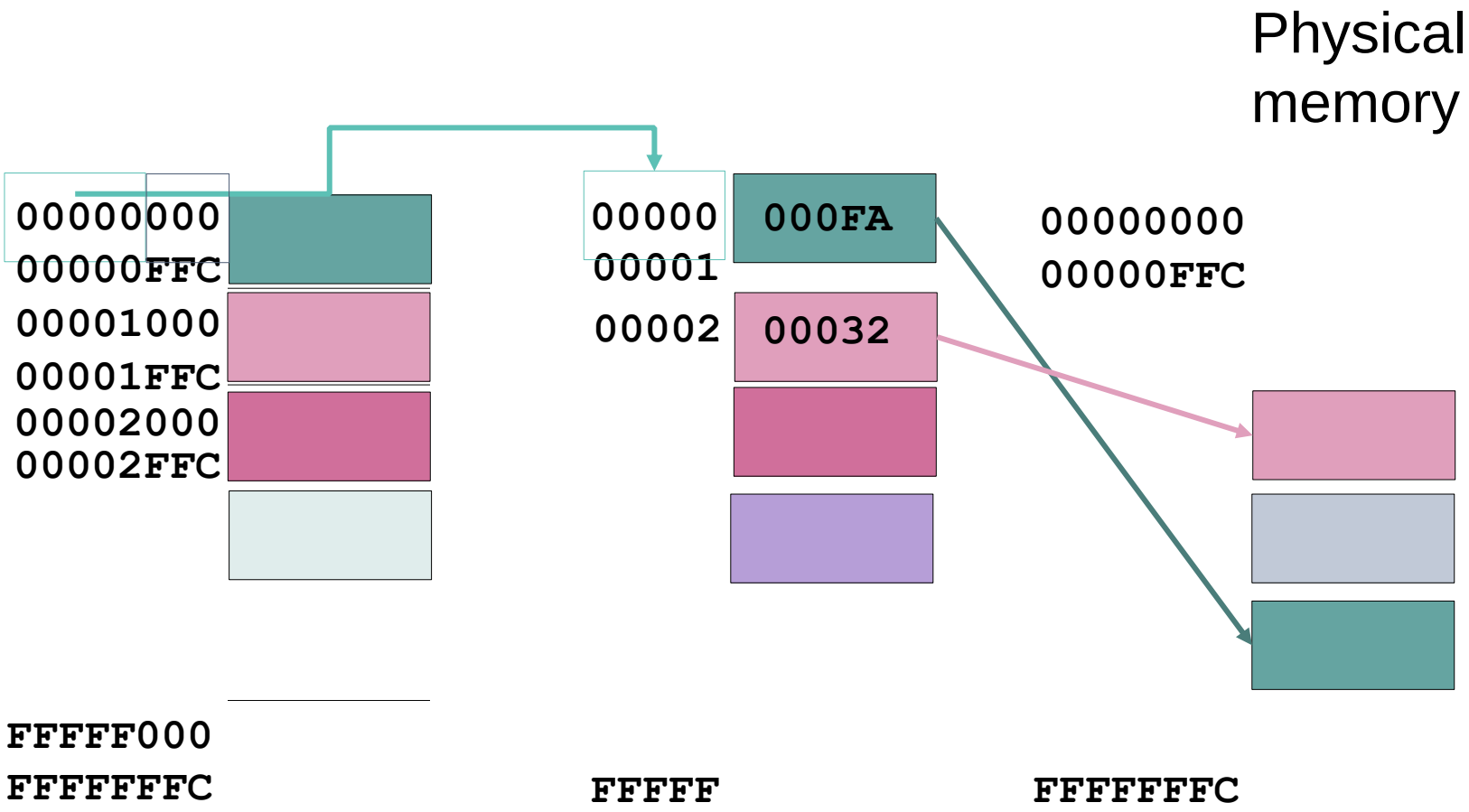
Høyskolen Kristiania



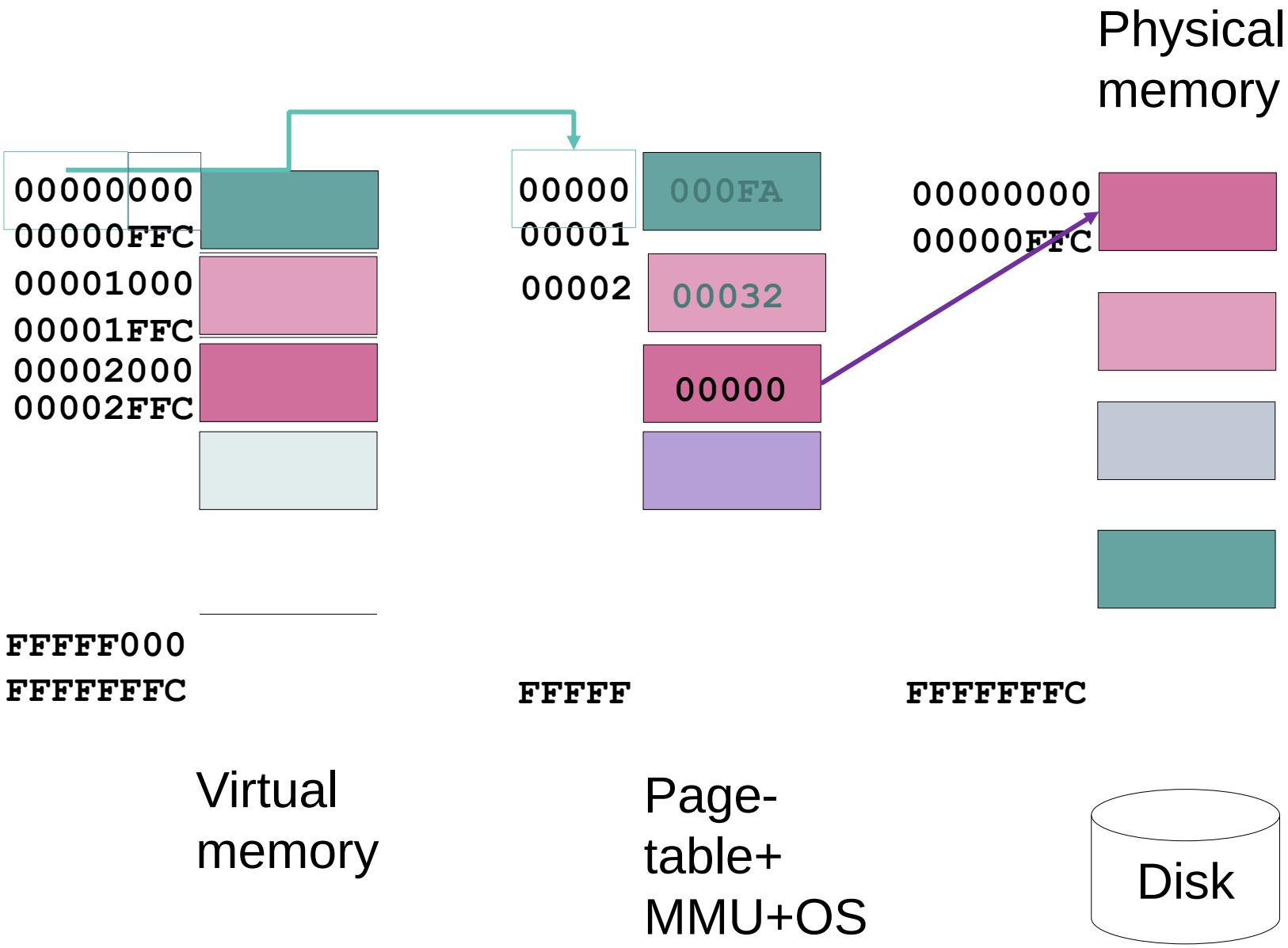
Page size = 0x1000 = 4096 bytes (dec)

Paging – virtual to physical memory #4

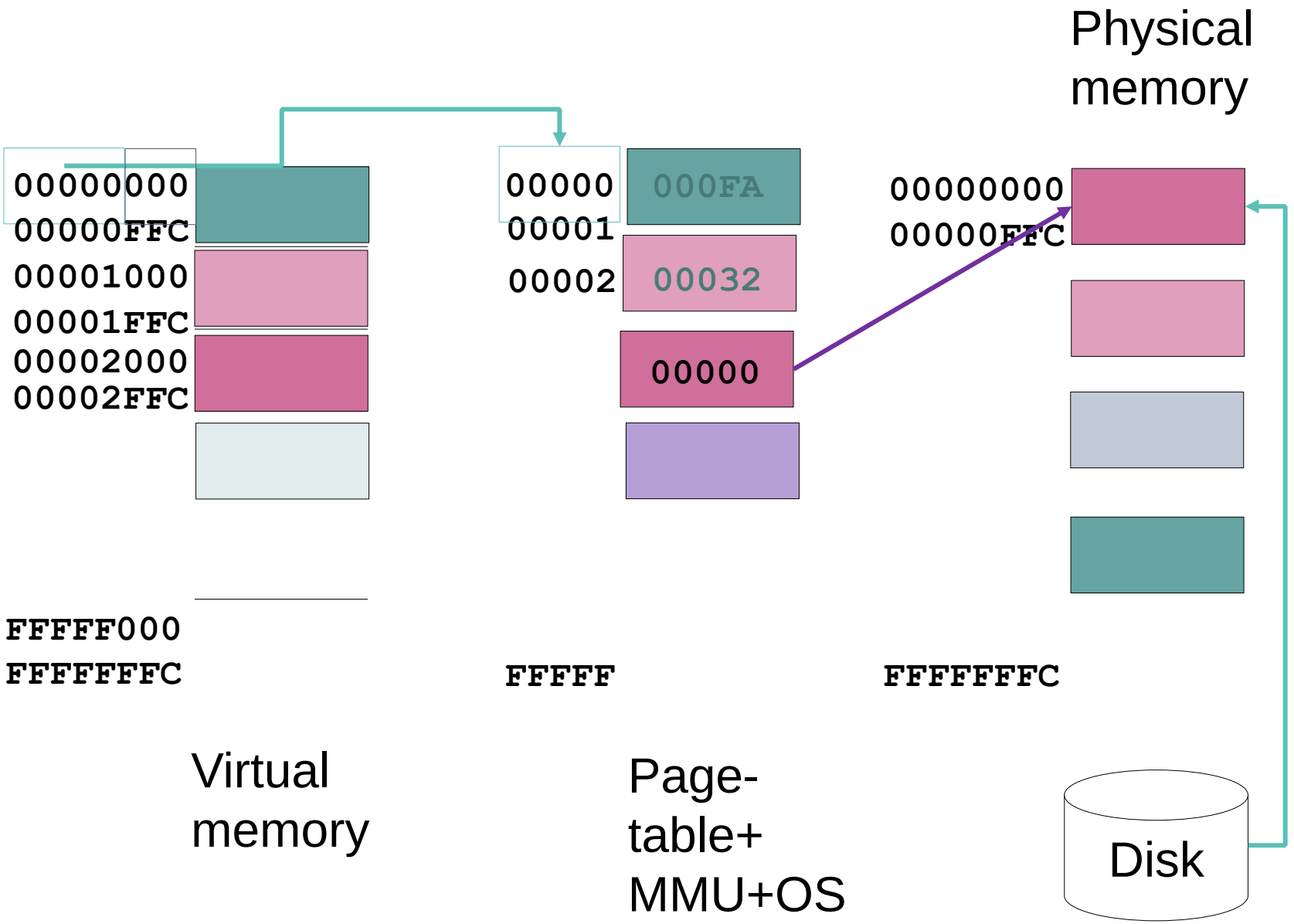
Høyskolen Kristiania



Paging – virtual to physical memory #5



Paging – virtual to physical memory #6



See also…

- https://en.wikipedia.org/wiki/Virtual_memory

(Virtual) memory types available to C programs

- Static memory
- Stack
- Heap

Static memory

- Allocated at start, deallocated at exit.
- Initialized to 0

```
#include <stdio.h>
int iNum, iCount = 0;
int iData[100];

main (void) {}
```

Stack

- Managed automatically (built into the CPU)
- push/pop
- Local to scope – cleared after scope (except for static)
- All *locally* declared variables go on stack (except for static)
- Not initialized automatically.

This is important, locally declared variables are not initialized to 0 automatically – you have to set it manually in the code. It can LOOK like it is set to 0 at first, but when stack memory is reused it will keep its value!

Global | static | (local == stack)

```
#include <stdio.h>
int i = 3, j;

int MyFunc (int par) {
    static int k = 42;
    k += 1;
    return (par + k);
}

void main (void) {
    int a, b;
    a = MyFunc(i);
    b = MyFunc(j);
}
```

a = 42 + 1 + 3

b = 42 + 1 + 1 + 0

i : global static, initialized to 3

j : global static, un-initialized
(so it is set to 0…)

k : static, initialized to 42,
increases by 1 (from 42)
for each call…

a, b : local (stack variable),
un-initialized, will get
any value on the stack when
main is called, since main it is
often set to 0 – but might not…

Heap – dynamic memory

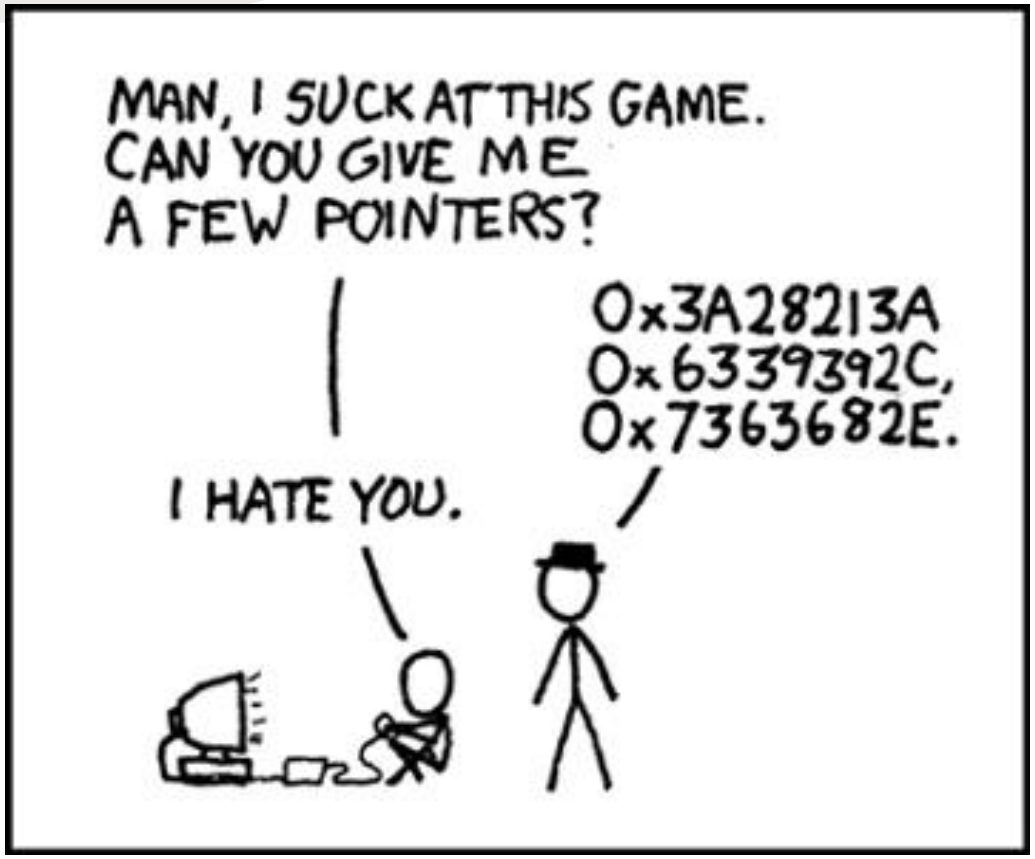
- Managed by Programmer
- Explicit memory allocation
- Not initialized.
- Will cause problems unless you know what you're doing…

sizeof()

- Finds the size of the operand in bytes
- Pointers always return size of the pointer itself
- Otherwise, it returns allocated space

A pointer

- Is simply the ***location*** of a variable in memory !



xkcd.com

Pointers – what for?

- Access the *data* or *variable* at the location

```
int main (void) {
    int a = 1, b = 2;
    int *p = &a;
    return b + *p;
}
```

Pointer declaration

- A pointer is also a variable that needs to be declared:

```
int *pInt;
float *pFloat;
char *pChar;
```

- Decides what type of variable this pointer is pointing to and hence the *size* of memory pointed to.

Notice the "missing" * in the first line...

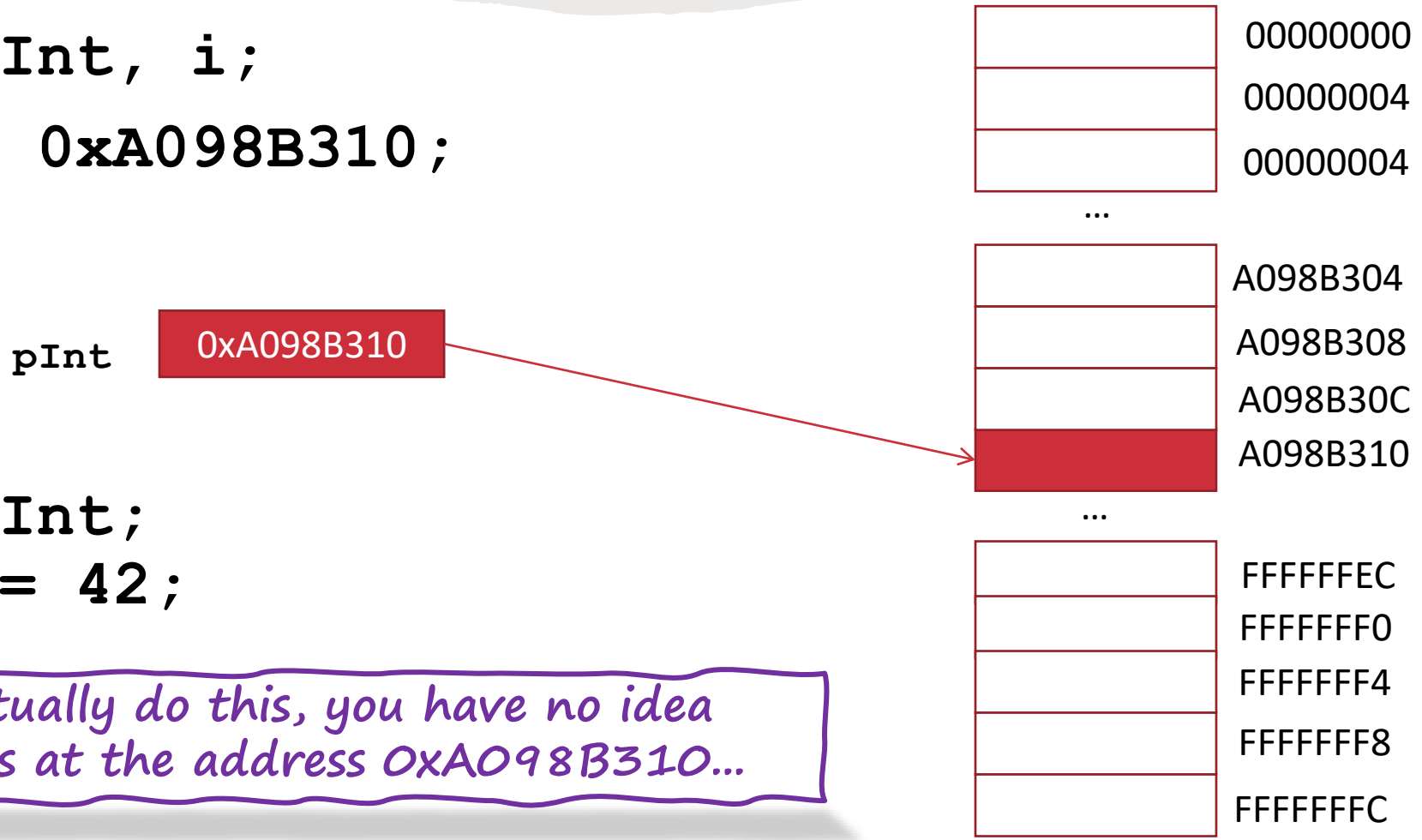
```
int *pInt, aVal;  // will declare a pointer and an integer.
int *pInt, *aVal; // will declare two pointers.
```

Pointer Declaration

- int *pInt;

Pointer Assignment

```
int *pInt, i;
pInt = 0xA098B310;
```



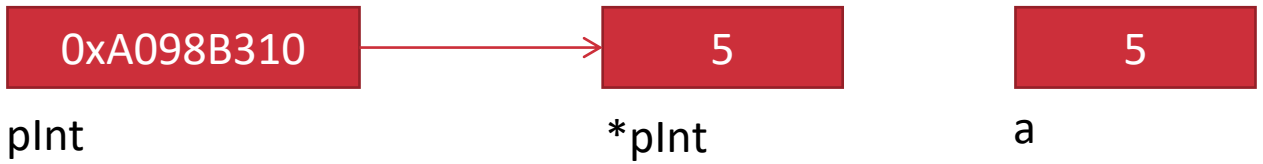

```
i = *pInt;
*pInt = 42;
```

Don't actually do this, you have no idea what lays at the address 0xA098B310...

Accessing memory by *pointer…

- Getting the value of the memory location pointed to is also called dereferencing the pointer:

```
int a;
*pInt = 5;
a = *pInt;
```



Address of a variable

- All variables have addresses
- Exceptions :
  - constants/literals
  - preprocessor defines
  - expressions unless resulting in a variable
  - registers
- ‘&’ – unary operator – different from bitwise operator
- Address can be requested for all variables simply by using ‘&’

```
int a = 4;
double d = 5;
printf(" %p and %p \n", &a, &d);
```

Address of a variable

- '&' on registers/constants – compile time error
- Address for a variable of type *t* has type *t* *

```
int a = 6;
int *pa;
pa = &a;
printf ("a=%d *pa=%d\n", a, *pa);

float b = 7.0;
float *pb;
pb = &b;
```

Use of pointers

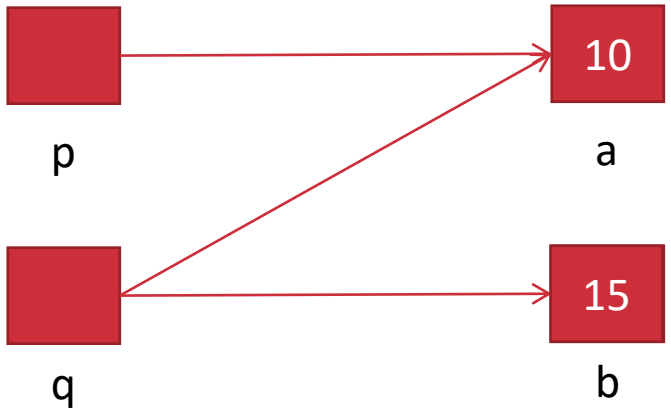
```
int *p, *q;
int a = 10;
int b = 25;
p = &a;
q = &b;
*q = 15;
q = p;
```

a = 10
b = 25

*q = (contents of address of b) = b = 15

pointer q = pointer p
Does not actually change neither a or b!

Use of pointers




```
int *p, *q;
int a = 10;
int b = 25;
p = &a;
q = &b;
*q = 15;
q = p;
```

*p? *q?

So…
*p = 10
q = p so *q = 10…

Types of Pointers

- Decides what type is being placed at the memory that a pointer points to. Also decides the size

- Example :

```
int *pInt;
double *pDouble;
void *pVoid;
int **pPointer;
```

Null Pointer

- NULL – preprocessor define for ‘0’
- Useful for memory management
- Initialize all pointers to NULL at declaration
  - Not required ,but good practice

Important; Good coding practice:
1) Initialize pointers to NULL
2) Check for NULL pointers

Improper (and unsafe) use of pointers deduct points on an exam, it is bad coding practice…

- Usually C does no default initialization:

```
int *p = NULL, *q = NULL, *r = NULL, *s = NULL;
if (s != NULL) {*s = 42;}
printf("%p - %p - %p - %p", p, q, r, s);
```

Høyskolen Kristiania

“Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end”

Henry Spencer’s 10 commandments for C-programmers.
https://www.seebs.net/c/10com.html

void *

- Generic pointer - *typeless*
- Cannot dereference
- Need casting before dereference

Pointer Assignment

- Pointer to stack memory – by using address operator
- Pointer to heap memory must be first allocated

```
int a = 1;
int *p = &a;
```

p assigned to address of a using the "address operator" - &

Heap memory: Memory allocation

- Allocate memory from heap
- Performed at runtime
- Not de-allocated automatically
- Useful for dynamic memory usage

malloc()

- man malloc
- malloc(size_t Size) – argument in bytes
- Usage:
  - malloc(sizeof(int))
  - malloc(10*sizeof(int))
  - malloc(20*sizeof(double))
- Returns (void *) always
- Returns NULL if failed
- Good code will check for output

Important; Good coding practice:
1) Check for NULL pointers (return value from malloc)
2) If out of memory malloc will fail, accessing this pointer crashes the program!

Improper (and unsafe) use of pointers deduct points on an exam, it is bad coding practice…

calloc()

- malloc and initialization to zero
- calloc(size_t Count, size_t Size)
- Requires number of elements along with size of each
- Usage:
  - calloc(10, sizeof(int));
  - calloc(1, sizeof(int));
  - calloc(100, sizeof(char));//what happens here?
- Returns (void *)
- Returns NULL for failure
- Checks for integer overflow of arguments

I no longer remember why, but I learned to use malloc rather than calloc, if it was simply for readability or if it was a performance issue I do not know, but I still only use malloc :-)

free()

- De-allocates memory
- **`free(void *p)`**
- Usage:

```
int *p;
p = (int *)malloc(10*sizeof(int));
free(p);
p = NULL;
```

Important; Good coding practice:
1) Always free allocated memory (explicitly)
2) Even though memory is freed when program ends such code could later be reused in other parts of the code where it might be called in a loop…

Memory leaks deduct points on an exam, it is bad coding practice…

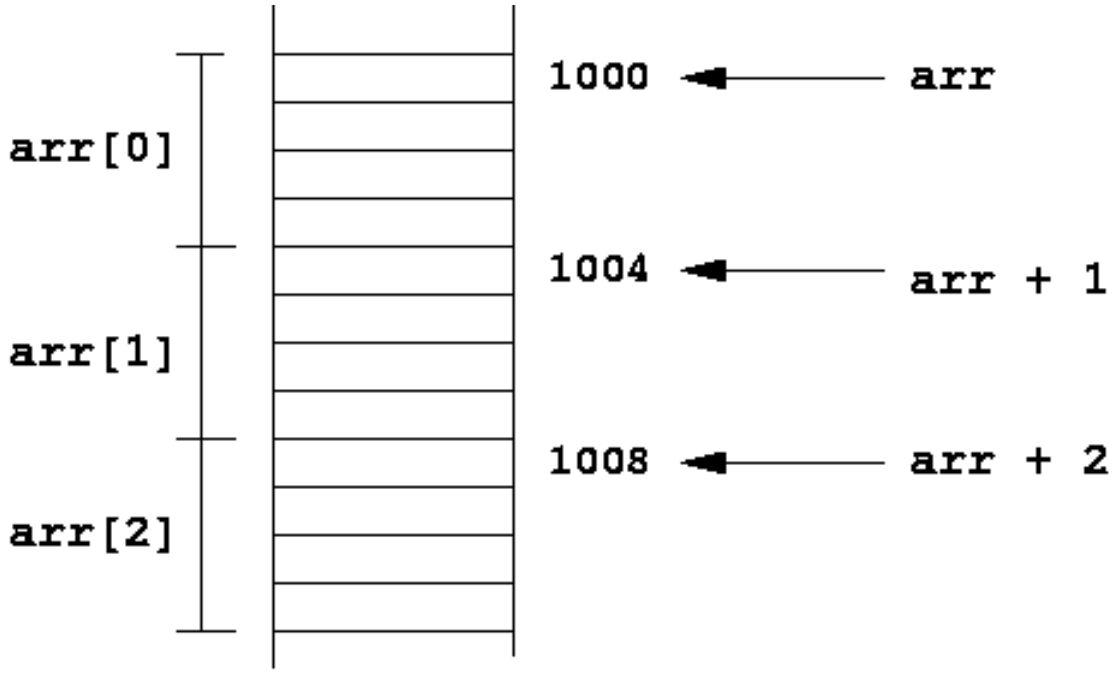
- p is not changed – but the memory is not available anymore
- Good idea to change p to NULL.

Pointer Arithmetic

- Ways to manipulate pointers
- Since pointers are just numbers – anything is possible
- What makes sense?
  - Addition of integer
  - subtraction of integers
  - Comparison operators
  - difference of two pointers

p + int

- Access elements of memory block after the pointer
- **`p+i`** is the same address as **`p[i]`**
- **`(p+i) == &p[i]`**
- **`*(p+i) == p[i]`**
- The addition depends on the type



p - int

- Access elements before a pointer
  - The location *must* be in the *same* memory block

- The size of memory offsets depends on type
- Useful for certain cases – stack
- Careful :
  - ‘off by one’ error

Comparison Operator

- <, <= , >, >=, ==, != between two pointers
- Useful for memory-block traversal
- Both must be inside the *specific* memory block
- Only pointers are compared not the value at the variables
- Example :

```
int *p,*q;
int pVal = 10, qVal = 15;
p = &pVal; q = &qVal;
(p > q) and (*p > *q) are different
```

p - q

- Difference operator
- Used to count the objects between two pointers
- Return type is ptrdiff_t defined in stddef.h
- Example :
  - p is a double array of size n with 0.0 somewhere, q is a double *

```
int position;
for(q=p; q<p+n;q++)
      if(*q == 0.0)
            break;
position = (q == p+n) ? -1 : q-(p+1)
```

Casting

- Implicit casting from/to (void *)
- Explicit casting is allowed
- Example:

```
int *pInt;
char *pChar;
pChar = (char *) pInt;
pInt = (int *)pChar;
```

- Problems with explicit casting?
- Avoid unless you know what you are doing!

```
Int x = 42;
Char *pPtr = NULL;

..

Void *pGeneric = NULL;
pGeneric = (char*)&x;
```

*Unable to cast from int * to char *, explisit cast required.*

Casting

- Example :

```
int *pInt, expr;
char *pChar;
pChar = (char *)pInt;
expr =(int *)(pChar + offSet) == pInt + 3;
```

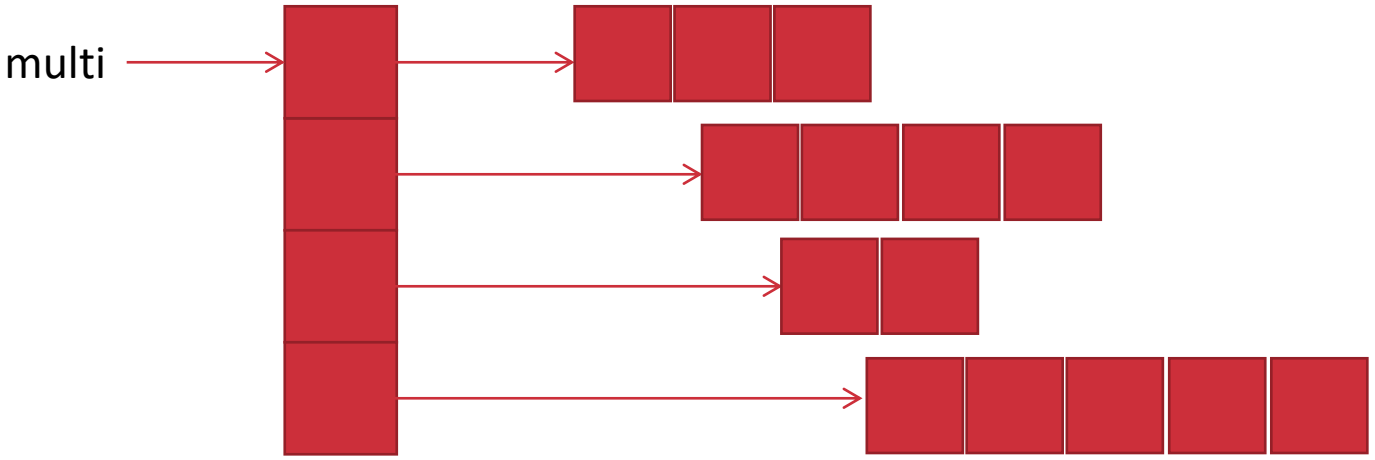
- What would you guess the offSet is so that the expr is true?

- Answer : offSet = 12

Memory operations (functions)

- **void *memcpy(void *dest, const void *src, size_t len)**
  - copy from one buffer to other
  - Both must be valid memories
  - Cannot have overlapping buffers
- **void *memmove(void *dest, const void *src, size_t len)**
  - Makes a copy first and then copy to dest
  - Handles overlap
- **void *memcmp(const void *p, const void *q, size_t len)**
  - lexicographic comparison
- **void *memchr(const void *p, int c, size_t len)**
  - find first occurence (char)c in p
  - memrchr – find last occurence

Pointers to pointers



int argc, char **argv

Argv[0] = “./hello”
Argv[1] = “parameter en”

Pointers with functions

- Pointers can be passed as function arguments or expected as return type.

```
void bsort(int *arr, int arrsize)
```

- Example:

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

- Be careful with returning pointers from local scope!
- More in later lectures!

Probable problems

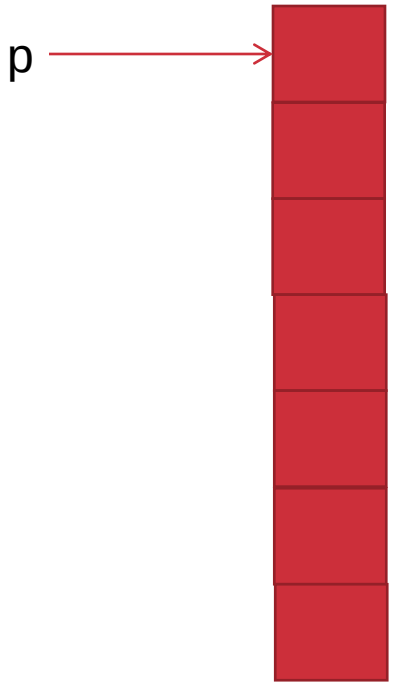
- Dangling reference
- Fragmentation
- Memory leaks
- Buffer overflow

Dangling Reference

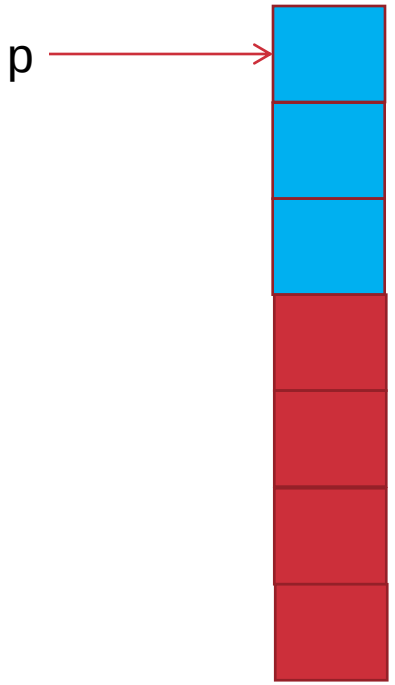
```
char *pChar = malloc(SOME_SIZE);
free(pChar);
```

- Now pChar is a dangling reference

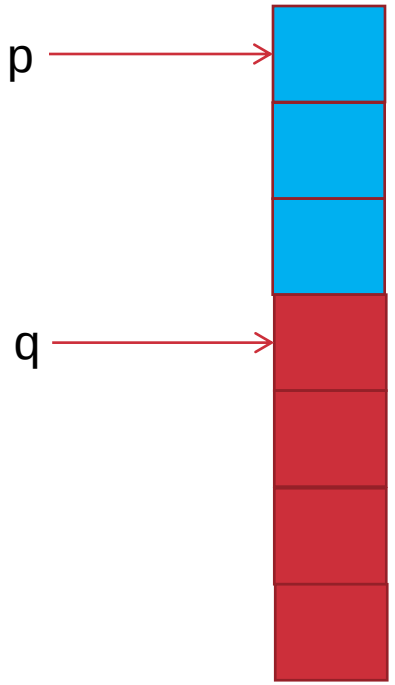
Fragmentation



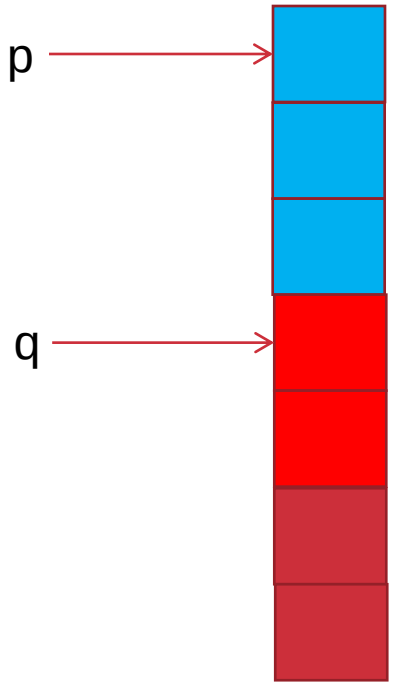
Fragmentation



Fragmentation

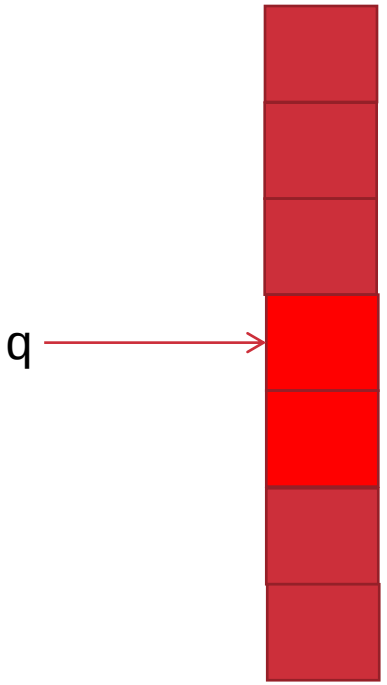


Fragmentation



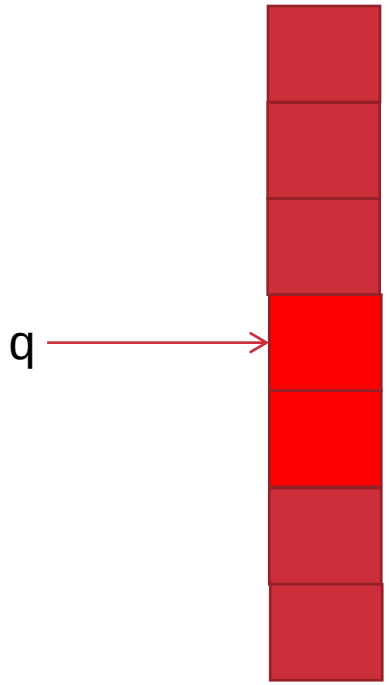
Fragmentation

Høyskolen Kristiania



Fragmentation

Høyskolen Kristiania



How much free space?

Memory Leaks

- free()
- Losing pointers
- Reassigning pointers
- Use valgrind!

Important; Good coding practice:
1) Always free allocated memory (explicitly)
2) The most common memory leak is simply NOT calling free…

Memory leaks deduct points on an exam, it is bad coding practice…

Buffer overflow

- C has no memory sizes accessible !
- *Example*:

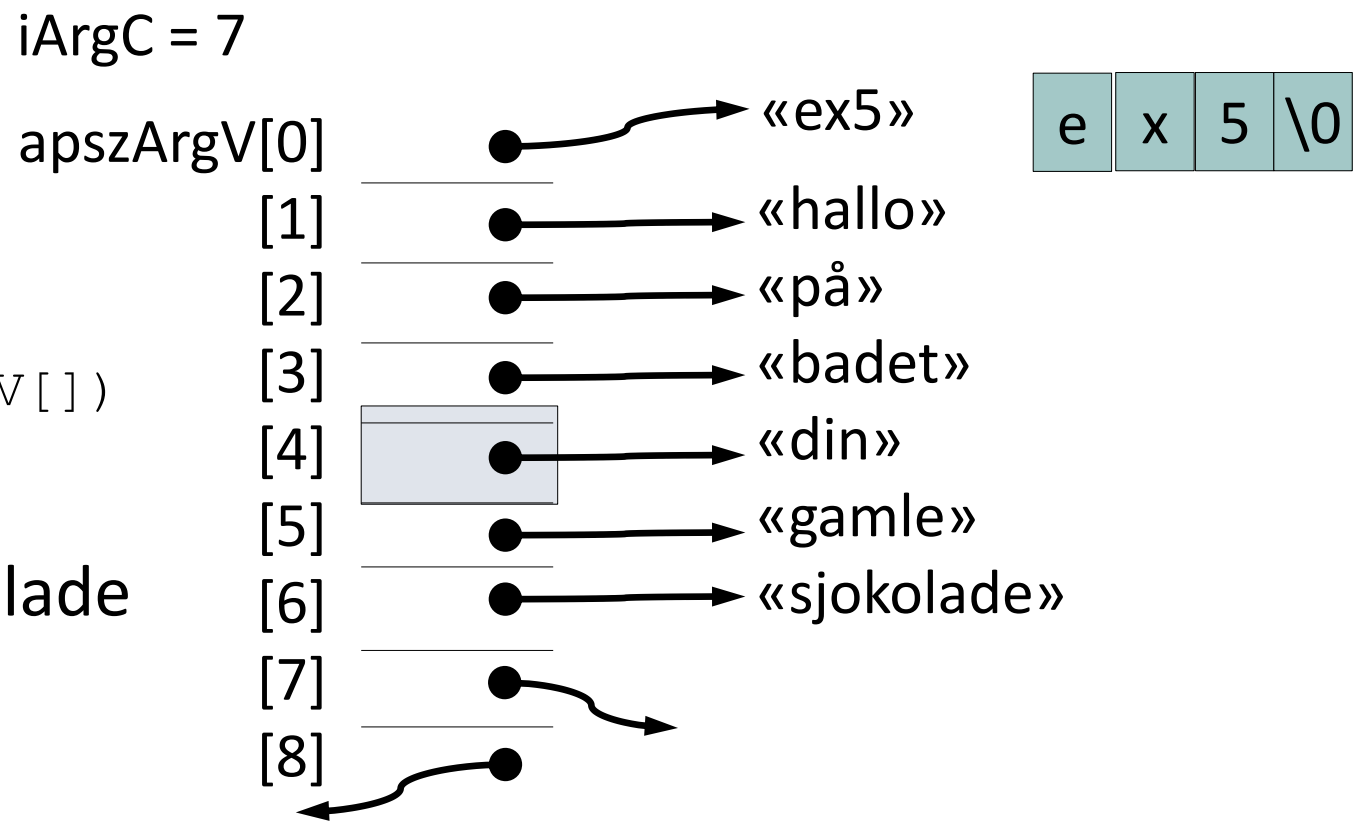
```
#define INTSIZE 10
int pInt[INTSIZE];
int pShort[2];

pShort[0] = 100;
pShort[1] = 200;
for(int i=0; i <= INTSIZE; i++){
        pInt[i] = i;
}
```

- What happened to the poor pShort?

argc, argv[] !!

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

$ ex5 hallo på badet din gamle sjokolade



Summary

- Pointers are awesome!
- Dynamic memory operations
  - malloc
  - calloc
  - free
- Type casting
- Pointers to pointers
- Function arguments
- Probable problems!

A hint about exam format

- There will only be 1 exam in this course, and it is a practical programming exam :-)

- Exam submission should be:
  - 1x PDF with a short documentation of the different a
  - 1x ZIP (or tar.gz) file that includes all source code

Due to the emergence of efficient AI programs, I have changed the exam format slightly from previous years – I think the result was even better and more fun exams, details will follow in a few weeks :-)

- All assignments should be buildable with a MAKEFILE (!)
- A demand for your exam is that I only want to write "make"…
- Compiler to be used must be gcc, no 3rd party tools or libraries
- Code must have comments, and follow "Best Practice" coding guidelines

Høyskolen Kristiania

Makefile – the one you should use

DO NOT COPY TEXT FROM SLIDES, MANY CHARACTERS ARE REPLACED BY POWERPOINT TO BE MORE "READABLE" GLYPHS, YOU NEED TO TYPE THE CHARACTERS IN YOUR EDITOR...

TAKES TOO LONG? LEARN THE "TOUCH-METHOD" FOR TYPING ON A KEYBOARD :-)

The following slide shows you a makefile you can use for all subsequent tasks and exercises – and will be required on the exam.

Use this makefile

Høyskolen Kristiania

Create these two sub folders

```
# Makefile template

INCLDIR = ./include
CC = gcc
CFLAGS = -O2
CFLAGS += -I$(INCLDIR)

OBJDIR = obj

_DEPS = source.h
DEPS = $(patsubst %,$(INCLDIR)/%,$(_DEPS))

_OBJS = source.o
OBJS = $(patsubst %,$(OBJDIR)/%,$(_OBJS))


$(OBJDIR)/%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)

hello: $(OBJS)
        gcc -o $@ $^ $(CFLAGS)

.PHONY: clean
clean:
        rm -f $(OBJDIR)/*.o *~ core $(INCLDIR)/*~
```

List all header files here

List all source files here, but given their .o file convention

Change output filename here

Høyskolen Kristiania