IN3200/IN4200: High-Performance Computing & Numerical Projects

Course overview & a crash course on C programming

Xing Cai

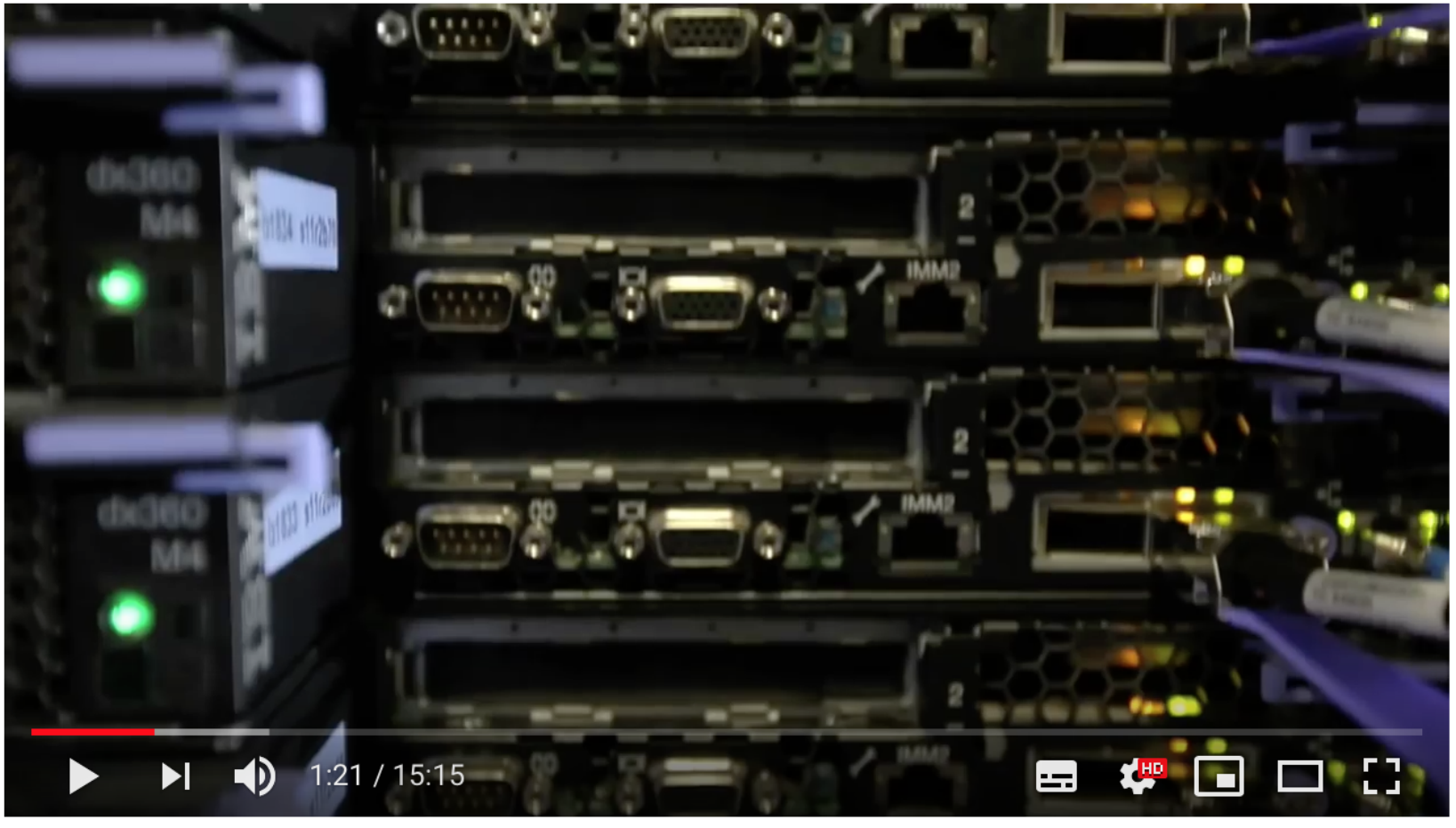
University of Oslo & Simula Research Laboratory, Norway

Spring 2024

Motivations for HPC

- Many problems in science & technology can benefit from large-scale or huge-scale computations
  - more details
  - better accuracy
  - more advanced models
- The need for computing is ever-increasing
- Therefore, **high-performance computing (HPC)** is required

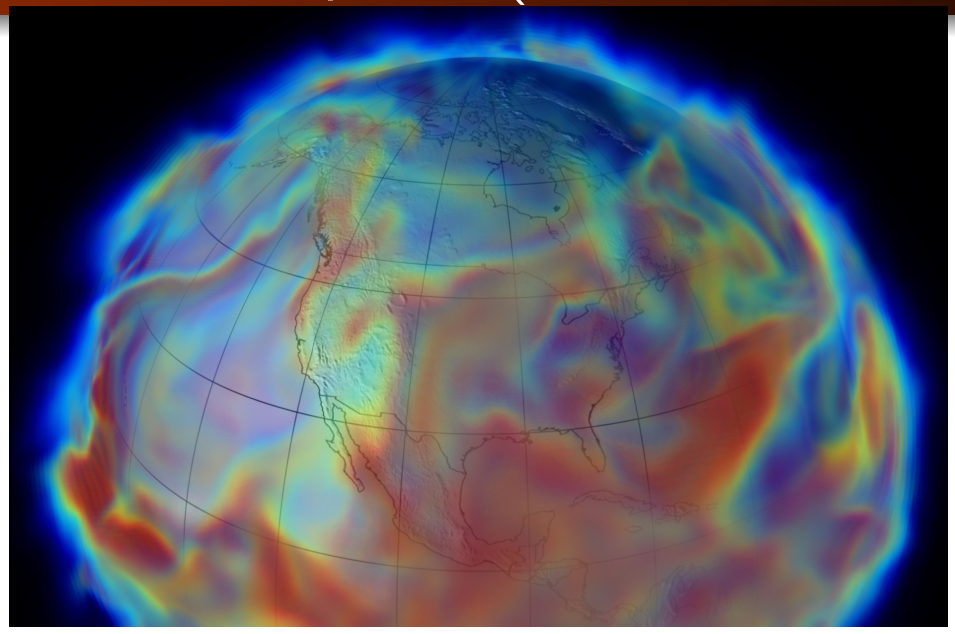
A YouTube video about **SuperComputing** (an extreme form of HPC)



Supercomputing and eScience (Eng)

https://www.youtube.com/watch?v=S9YPcPtPsuY

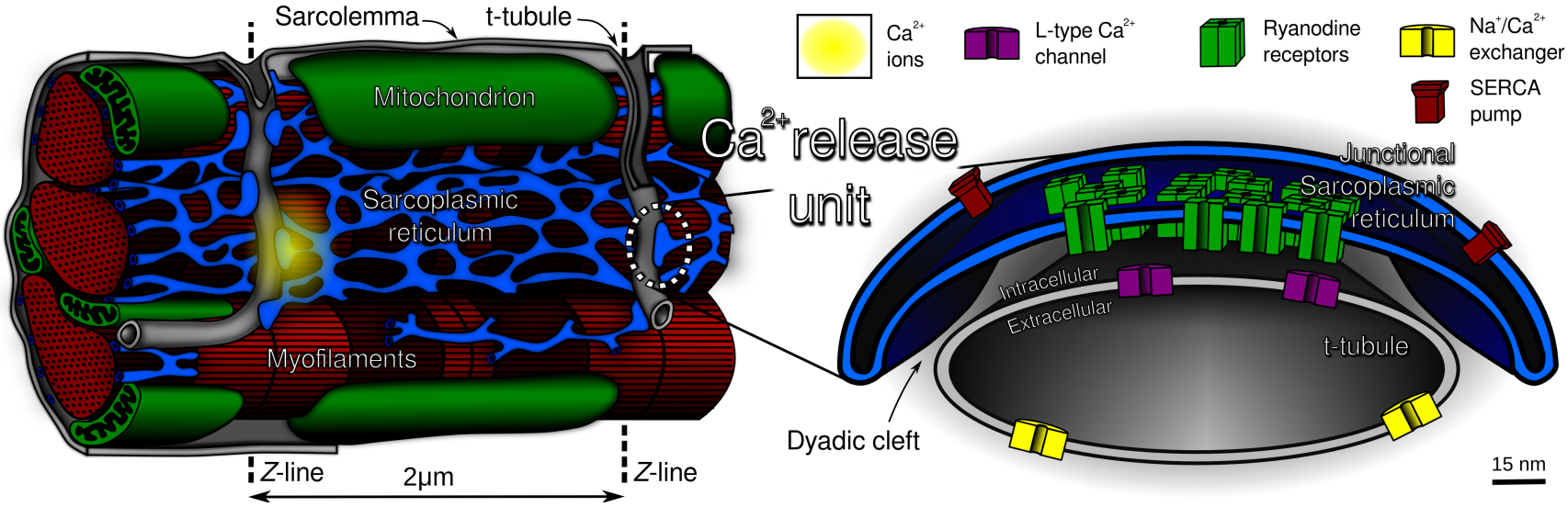
Huge computation example 1 (Climate Simulation)



NASA Center for Climate Simulation

- Earth surface area: $510,072,000\ \text{km}^2$
- If a spatial resolution of $1 \times 1\text{km}^2$ is adopted $\rightarrow 5.1 \times 10^8$ (510 million) small patches
- If a spatial resolution $100 \times 100\text{m}^2$ is adopted $\rightarrow 5.1 \times 10^{10}$ (51 billion) small patches
- Additional layers in the vertical direction
- High resolution in the time direction

Example 2 (Subcellular Calcium Dynamics Simulation)



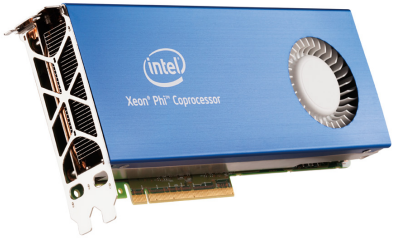
- Size of one cardiac muscle cell: $100\mu\text{m} \times 10\mu\text{m} \times 10\mu\text{m}$
- Width of calcium release channels: 1 nanomater (nm)
- Ideal computational mesh resolution: 1 nm
- Computational mesh required: $10^5 \times 10^4 \times 10^4$ (in total 10^{13} computational voxels)
- Number of simulation time steps needed: $\sim 10^6$

Motivations (cont'd)

- Parallel computers are now everywhere!
  - CPUs nowadays have multiple "cores" on a chip
  - One computer may have several multicore chips
  - There are also accelerator-based parallel architectures — GPGPU (general-purpose graphics processing unit)
  - Clusters of different kinds







What do we learn in IN3200/IN4200?

High-performance computing (HPC) – an introduction

- Proper implementation of numerical algorithms
- Effective use of the hardware for numerical computations

After finishing the course, you should

- be able to write simple parallel programs with sufficiently good performance
- be able to learn more about advanced computing later on your own

Part 1 of the course: Serial programming

- A brief architectural overview of modern cache-based microprocessors
- Inherent performance limitations of microprocessors
- Basic C programming
- Optimization strategies of serial code

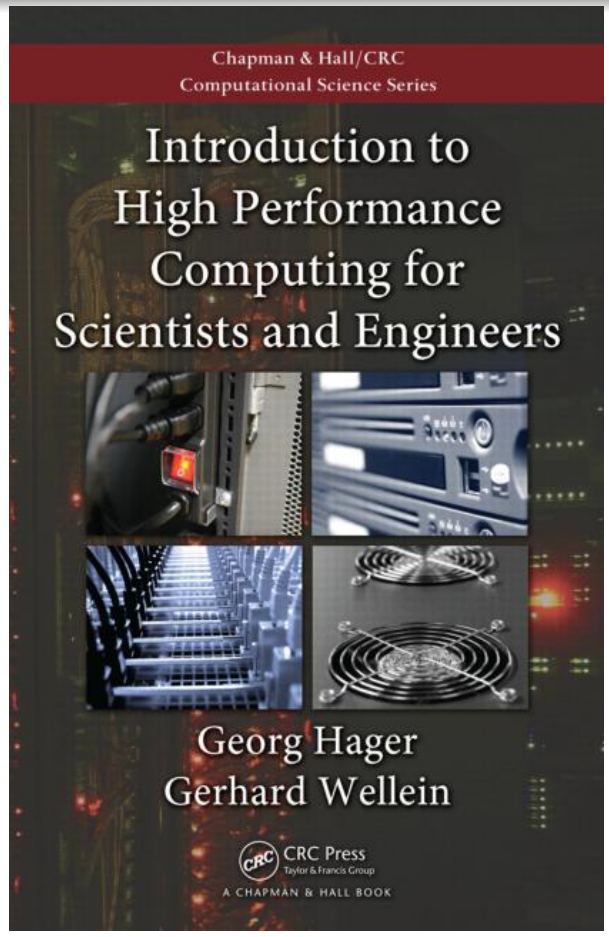
Part 2 of the course: Parallel programming

- Parallel computer architecture
- Theoretical considerations of parallel computing
- Shared-memory parallel programming (OpenMP)
- Distributed-memory parallel programming (MPI)

Why learning parallel programming?

- Parallel computing – a form of parallel processing by concurrently utilizing multiple computing units for one computational problem
  - shortening computing time
  - solving larger problems
- However ...
  - modern multicore-based computers are good at multi-tasking, but not good at automatically computing one problem in parallel
  - automatic parallelization compilers have had little success
  - special parallel programming languages have had little success
  - serial computer programs have to be modified or completely rewritten to utilize parallel computers
- Learning parallel programming is thus important!

Textbook



Georg Hager, Gerhard Wellein
Introduction to High Performance Computing for Scientists and Engineers

1st Edition, CRC Press, ISBN 9781439811924

Teaching approaches

- Focus on fundamental issues
  - parallel programming = serial programming + finding parallelism + enforcing work division and collaboration
- Use of examples relevant for natural sciences
  - mathematical details are not required
  - understanding basic numerical algorithms is needed
  - implementing basic numerical algorithms is essential
- Hands-on programming exercises and tutoring
- English is the "official language" of the course, but students should feel free to ask questions, write emails/reports in Norwegian

What is serial programming?

- Roughly speaking, a computer program executes a sequence of operations applied to data structures
- A program is normally written in a programming language
- Data structures:
  - variables of primitive data types (`char`, `int`, `float`, `double` etc.)
  - variables of composite and abstract data types (`struct` in C, `class` in Java & Python)
  - array variables
- Operations:
  - statements and expressions
  - functions

Similarities and differences between languages

- For scientific applications, arrays of numerical values are the most important basic building blocks of data structure
- Extensive use of `for`-loops for doing computations
- Different syntax details
  - allocation and deallocation of arrays
    - Java: `double[] v=new double[n];`
    - C: `double *v=malloc(n*sizeof(double));`
    - Python: `v=zeros(n,dtype=float64)` (using NumPy)
  - definition of composite and abstract data types
  - I/O

C as the main choice of programming language

- C is one of the dominant programming languages in computational sciences
- Syntax of C has inspired many newer languages (C++, Java, Python)
- Good computational efficiency
- C is ideal for using MPI and OpenMP (also GPU programming)
- We will thus choose C as the main programming language
- (Most of the textbook's coding examples are in Fortran, but many of the "performance-engineering" principles are the same.)

A crash course on C programming

For a more "complete" tutorial on C programming:
https://www.tutorialspoint.com/cprogramming/

First things first

- A program in C is made up of
  - Preprocessor commands
  - Variables
  - Statements and expressions
  - Functions
  - Comments
- A program in C can be as simple as having only 3 lines, or as comprehensive as being composed of millions of lines
- A program in C can be stored in one file with name extension `.c` (or spread over many `.h` and `.c` files)
- Use of libraries—groups of already-coded functions and declarations—actually happens all the time

Identifiers

- An identifier is a name used to identify a variable, function, or any other user-defined item
- Examples of acceptable identifiers

```
mohd        zara    abc    move_name   a_123
myname50    _temp   j      a23b9       retVal
```

- C is a case-sensitive programming language

Keywords – reserved words

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Keywords can not be used as identifiers.

C data types

- **Basic Types**
  They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types
- **Derived types**
  They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types

Integer types

Type	Storage size	Value range
`char`	1 byte	-128 to 127 or 0 to 255
`unsigned char`	1 byte	0 to 255
`signed char`	1 byte	-128 to 127
`int`	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
`unsigned int`	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
`short`	2 bytes	-32,768 to 32,767
`unsigned short`	2 bytes	0 to 65,535
`long`	4 bytes	-2,147,483,648 to 2,147,483,647
`unsigned long`	4 bytes	0 to 4,294,967,295

The `sizeof` operator

To get the exact size of a type or a variable on a particular platform, you can use the `sizeof` operator. The expression `sizeof(type)` yields the storage size of the object or type in number of bytes.

```
#include <stdio.h>

int main() {
   printf("Storage size for int : %d \n", sizeof(int));

   return 0;
}
```

Floating-point types

Type	Storage size	Value range	Precision
`float`	4 bytes	1.2E-38 to 3.4E+38	6 decimal places
`double`	8 bytes	2.2E-308 to 1.8E+308	15 decimal places
`long double`	16 bytes	3.4E-4932 to 1.2E+4932	18 decimal places

Note: The actual values can be machine-dependent!

C variables

A variable is a name given to a storage area that a C program can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive.

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

C operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators:

- Arithmetic operators `+` `-` `*` `/` `%` `++` `--`
- Relational operators `==` `!=` `>` `<` `>=` `<=`
- Logical operators `&&` `||` `!`
- Bitwise operators
- Assignment operators

Loops

To execute a statement or a group of statements multiple times:

- `for`
- `while`
- `do ... while`

Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is `main()`, and you can define additional functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The general form of a function definition in C programming language:

```
return_type function_name( parameter list ) {
   body of the function
}
```

Function arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, the formal parameters get the values (that is, copies) of the **actual parameters**.

One example

```
#include<stdio.h>
void func_1(int);

int main()
{
    int x = 10;

    printf("Before function call\n");
    printf("x = %d\n", x);

    func_1(x);

    printf("After function call\n");
    printf("x = %d\n", x);

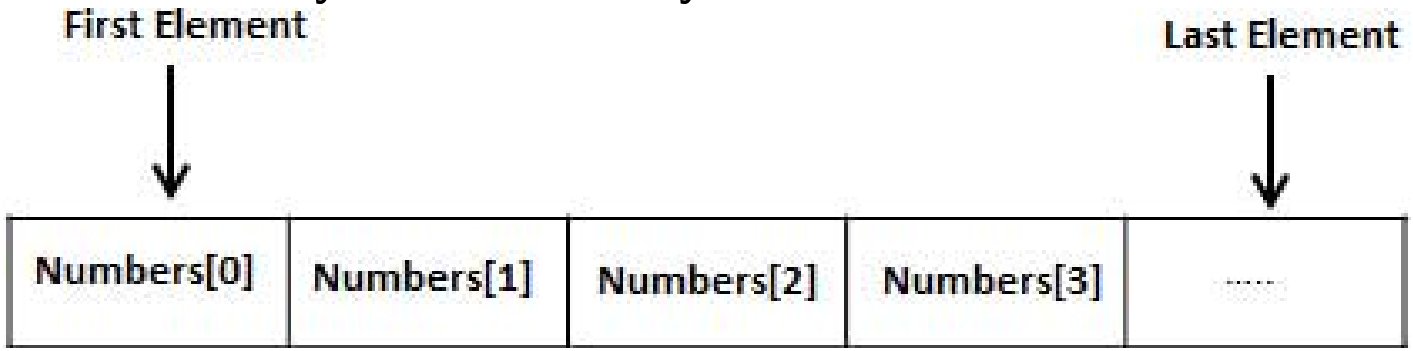
    return 0;
}

void func_1(int a)
{
    a += 1;
    a++;
    printf("\na = %d\n\n", a);
}
```

Arrays

An array is one kind of data structure that can store a sequential collection of elements of the same type.

Instead of declaring individual variables, such as `Number0`, `Number1`, ..., and `Number99`, you can declare one array variable, named such as `Numbers`, and use `Numbers[0]`, `Numbers[1]`, ..., and `Numbers[99]` to represent individual variables. A specific element in an array is accessed by an index.



Fix-sized arrays

A programmer can specify the type of the elements and the number of elements required by an array

```
type arrayName [ arraySize ];
```

`arraySize` must be an integer constant greater than zero.

However, long arrays are created in another way!
Also, very often a programmer may not know the exact array length.

Address in memory

Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory.

```
#include <stdio.h>

int main () {

   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1);
   printf("Address of var2 variable: %x\n", &var2);

   return 0;
}
```

Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer variable before using it to store any variable address.

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

How to use pointers?

- Define a pointer variable
- Assign the address of a variable to a pointer variable
- Access the value at the address stored in the pointer variable (via operator *)

```
#include <stdio.h>

int main () {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

More pointer concepts

- **Pointer arithmetic**: Four arithmetic operators can be used on pointers: ++, --, +, -
- **Array of pointers**: You can define an array to hold a sequence of pointers.
- **Pointer to pointer**: C allows you to have pointer on a pointer and so on.
- **Passing pointers to functions in C**: Passing an argument by address allows the passed argument to be changed.

Dynamic memory management

The C programming language provides several functions for memory allocation and management. These functions can be found in the `<stdlib.h>` header file.

- `void *calloc(int num, int size);` – allocates an array of num elements each of which size in bytes will be size.
- `void free(void *address);` – releases a block of memory block specified by address.
- `void *malloc(int num);` – allocates an array of num bytes and leave them uninitialized.
- `void *realloc(void *address, int newsize);` – re-allocates memory extending it upto newsize.

Example of dynamic memory allocation

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

int main()
{
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int));
    if(ptr == NULL) {
        printf("Error! memory not allocated."); exit(-1);
    }

    printf("Enter elements: ");
    for (i = 0; i < n; ++i) {
        scanf("%d", &(ptr[i]));
        sum += ptr[i];
    }

    printf("Sum = %d\n", sum);
    free(ptr);
    return 0;
}
```

Allocating multi-dimensional arrays

- We want a 2D array for representing a $m \times n$ matrix

$$\mathrm{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

- One way to allocate a 2D array:

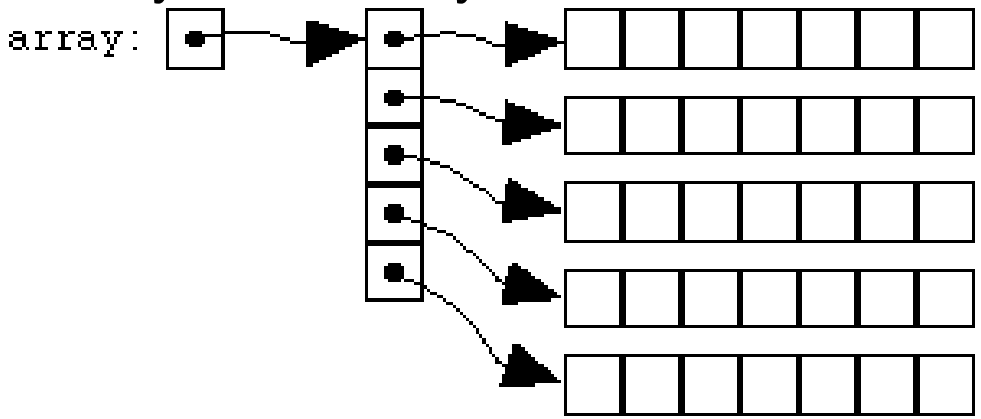
```
double **A = (double**)malloc(m*sizeof(double*));
for (i=0; i<m; i++)
   A[i] = (double*)malloc(n*sizeof(double));
```

- An example of usage:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    A[i][j] = i+j;
```

More about two-dimensional arrays in C (1)

- C doesn't have true multi-dimensional arrays, a 2D array is actually an array of 1D arrays



- `A[i]` is a pointer to row number $i+1$
- It is also possible to use static memory allocation of fix-sized 2D arrays, for example:

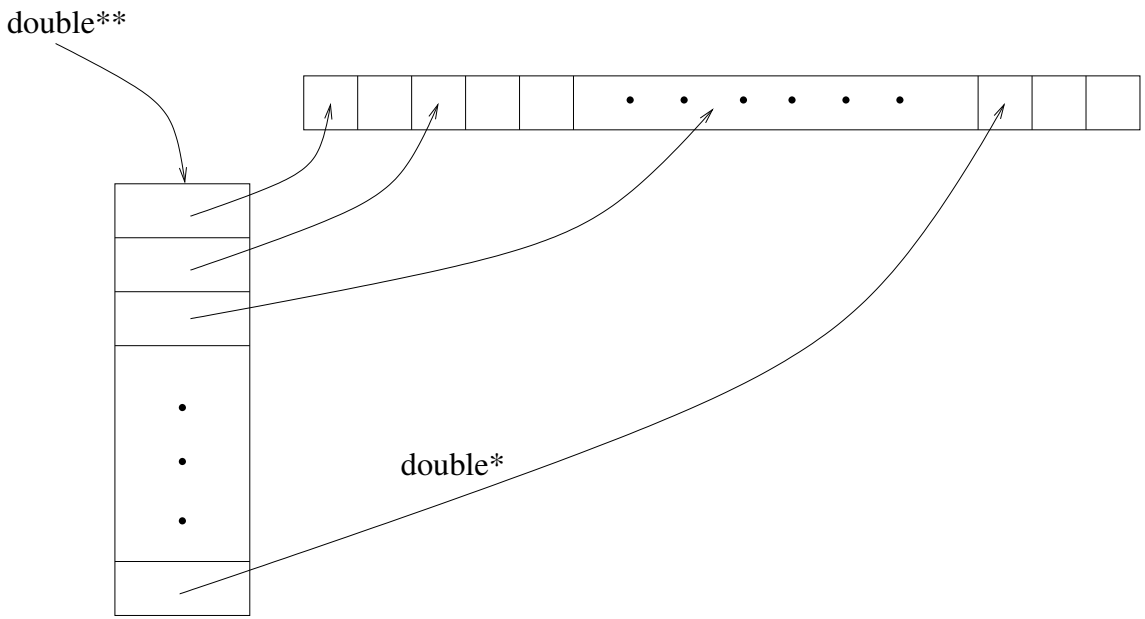
  ```
  double A[10][8];
  ```

  However, the size of the array is decided at compiler time (not runtime)

More about two-dimensional arrays in C (2)

- Another way of dynamic allocation, to ensure contiguous underlying data storage (for good use of cache):

```
double *A_storage=(double*)malloc(m*n*sizeof(double));
double **A = (double**)malloc(m*sizeof(double*));
for (i=0; i<m; i++)
  A[i] = &(A_storage[i*n]);
```



Deallocation of arrays in C

- If an array is dynamically allocated, it is important to free the storage when the array is not used any more
- Example 1

```
int *p = (int*)malloc(n*sizeof(int));
/* ... */
free(p);
```

- Example 2

```
double **A = (double**)malloc(m*sizeof(double*));
for (i=0; i<m; i++)
   A[i] = (double*)malloc(n*sizeof(double));
/* ... */
for (i=0; i<m; i++)
  free(A[i]);
free(A);
```

- Be careful! Memory allocation and deallocation can easily lead to errors

Beware of function arguments!

- **All arguments to a C function are passed by value**
- That is, a copy of each argument is passed to the function

```
void test (int i) {
  i = 10;
}
```

The change of `i` inside `test` has no effect when the function returns

- Passing pointers as function arguments can be used to get output

```
void test (int *i) {
  *i = 10;
}
```

The change of `i` inside `test` now has effect

Function example 1: swapping two values

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

Function example 2: smoothing a vector

- We want to smooth the values of a vector v by the following formula:

$$v_i^{\text{new}} = v_i + c\left(v_{i-1} - 2v_i + v_{i+1}\right), \quad 1 \leq i < n-1$$

where c is a constant

```
void smooth (double *v_new, double *v, int n, double c)
{
  int i;
  for (i=1; i<n-1; i++)
    v_new[i] = v[i] + c*(v[i-1]-2*v[i]+v[i+1]);
  v_new[0] = v[0];
  v_new[n-1] = v[n-1];
}
```

- Similar computations occur frequently in numerical computations

Function example 3: matrix-vector multiplication

- We want to compute y = Ax, where A is a $m \times n$ matrix, y is a vector of length m and x is a vector of length n:

$$y_i = A_{i1}x_1 + A_{i2}x_2 + \ldots A_{in}x_n = \sum_{j=1}^{n} A_{ij}x_j, \quad 1 \leq i \leq m$$

```
void mat_vec_prod (double **A, double *y, double *x,
                   int m, int n)
{
  int i,j;
  for (i=0; i<m; i++) {
    y[i] = 0.0;
    for (j=0; j<n; j++)
      y[i] += A[i][j]*x[j];
  }
}
```

Command-line arguments

Input arguments to the `main` function:

```
#include <stdio.h>

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

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```