Part III

Review of C

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

Example (Basic C program)

```c
#include <stdio.h>
// include stdio.h to print
int main() // always called first
{ // braces delimit code block
  printf("Hello world.\n");
  // function call to print
  // line ends with a semicolon
  return 0;
  // optional return value to shell
}
```

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

Example (Basic C program)

```c
#include <stdio.h>
// include stdio.h to print
int main() // always called first
{ // braces delimit code block
  printf("Hello world.\n");
  // function call to print
  // line ends with a semicolon
  return 0;
  // optional return value to shell
}
```

```
                                  ⎧-O2
                                  ⎪-Os
$ gcc -o hello hello.c -std=c99  ⎨-O3
                                  ⎩-Ofast
$
```

compute • calcul CANADA

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

Example (Basic C program)

```c
#include <stdio.h>
// include stdio.h to print
int main() // always called first
{ // braces delimit code block
  printf("Hello world.\n");
  // function call to print
  // line ends with a semicolon
  return 0;
  // optional return value to shell
}
```

```
                                     ⎧-O2
                                     ⎪-Os
$ gcc -o hello hello.c -std=c99     ⎨-O3
                                     ⎩-Ofast

$ ./hello
Hello world.
$
```

C review: Basics

- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- We will use C99.

Example (Basic C program)

```c
#include <stdio.h>
// include stdio.h to print
int main() // always called first
{ // braces delimit code block
  printf("Hello world.\n");
  // function call to print
  // line ends with a semicolon
  return 0;
  // optional return value to shell
}
```

```
                                      -O2
                                      -Os
$ gcc -o hello hello.c -std=c99  {    -O3
                                      -Ofast

$ ./hello
Hello world.
$ echo $?
0
$ █
```

C review: Functions

Function declaration (prototype)

```
returntype name(argument-spec);
```

Function definition

```
returntype name(argument-spec) {
  statements
}
```

Function call

```
var=name(arguments);
f(name(arguments));
```

Procedures

Procedures are functions with return-type `void` ; called without assignment.

C review: Variables

Define a variable with

```
type name [= value];
```

- *type* may be a
  - built-in type:
    - floating point type:
      `float`, `double`, `long double`
    - integer type:
      `short`,[`unsigned`] `int`, [`unsigned`] `long int` ,[`unsigned`] `long long int`
    - character or string of characters:
      `char`, `char*`
  - array, pointer
  - structure, enumerated type, union
- Variable declarations and code may be mixed in C99.
- Variables can be initialized to a *value* when declared.
  Any non-initialized variable is not set to zero, but has a random value!

C review: Loops

```
for (initialization; condition; increment) {
  statements
}
```

```
while (condition) {
  statements
}
```

You can use `break` to exit the loop.

C review: Loops

```
for (initialization; condition; increment) {
  statements
}
```

```
while (condition) {
  statements
}
```

You can use `break` to exit the loop.

Example

```
#include <stdio.h>
int main() {
  for (int i=1; i<=10; i++)
    printf("%d ",i);
  // note the omitted braces
  printf("\n");
}
```

C review: Loops

```
for (initialization; condition; increment) {
  statements
}
```

```
while (condition) {
  statements
}
```

You can use `break` to exit the loop.

Example

```c
#include <stdio.h>
int main() {
  for (int i=1; i<=10; i++)
    printf("%d ",i);
  // note the omitted braces
  printf("\n");
}
```

```
$ gcc -o count count.c -O2 -std=c99
$ ./count
1 2 3 4 5 5 6 7 8 9 10
$ █
```

C review: Pointers

```
type *name;
```

C review: Pointers

*type *name*;

Example (Pointer assignment)

```
#include <stdio.h>
int main() {
  int a=7,b=5;
  int *ptr=&a;
  a = 13;
  b = *ptr;
  printf("b=%d\n,b);
}
```

```
$ gcc -o ptrex ptrex.c -O2 -std=c99
$ ./ptrex
b=13
$ █
```

C review: Pointers

```
type *name;
```

Example (Pointer assignment)

```
#include <stdio.h>
int main() {
  int a=7,b=5;
  int *ptr=&a;
  a = 13;
  b = *ptr;
  printf("b=%d\n,b);
}
```

```
$ gcc -o ptrex ptrex.c -O2 -std=c99
$ ./ptrex
b=13
$ █
```

Example (Pass by reference)

```
void inc(int *i) { (*i)++; }
int main() {
  int j=10;
  inc(&j);
  return j;
}
```

```
$ gcc -o passref passref.c -O2 -std=c99
$ ./passref
$ echo $?
11
$ █
```

C review: Automatic arrays

```
type name[number];
```

- *name* is equivalent to a pointer to the first element.
- Usage `name[i]`. Equivalent to `*(name+i)`.
- C arrays are zero-based.

C review: Automatic arrays

```
type name[number];
```

- *name* is equivalent to a pointer to the first element.
- Usage *name*[i]. Equivalent to *(*name*+i).
- C arrays are zero-based.

Example

```c
#include <stdio.h>
int main() {
  int a[10]={1,2,3,4,5,6,7,8,9,11};
  int sum=0;
  for (int i=0; i<10; i++)
    sum += a[i];
  printf("sum=%d\n,sum);
}
```

```
$ gcc -o autoarr autoarr.c -O2 -std=c99
$ ./autoarr
56
$ █
```

C review: Automatic arrays

```
type name[number];
```

- *name* is equivalent to a pointer to the first element.
- Usage `name[i]`. Equivalent to `*(name+i)`.
- C arrays are zero-based.

Example

```c
#include <stdio.h>
int main() {
  int a[10]={1,2,3,4,5,6,7,8,9,11};
  int sum=0;
  for (int i=0; i<10; i++)
    sum += a[i];
  printf("sum=%d\n,sum);
}
```

```
$ gcc -o autoarr autoarr.c -O2 -std=c99
$ ./autoarr
56
$ █
```

Gotcha:

- There's an compiler dependent limit on *number*.
- C standard only says at least 65535 bytes.

C review: Dynamically allocated arrays

Requires header file:

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

Defined as a pointer to memory:

```
type *name;
```

Allocated by a function call:

```
name=malloc(number*sizeof(type));
```

Usages:

```
a=name[number];
```

Deallocated by a function call:

```
free(name);
```

- System function call can access all available memory.
- Can check if allocation failed ($name == 0$).
- Can control when memory is given back.

C review: Dynamically allocated arrays

Example

C review: Dynamically allocated arrays

Example

```c
#include <stdlib.h>
#include <stdio.h>
void printarr(int n, int *a) {
  for (int i=0;i<n;i++)
    printf("%d ", a[i]);
  printf("\n");
}
int main(){
  int n=100;
  int *b=malloc(n*sizeof(*b));
  for (int i=0;i<n;i++)
    b[i]=i*i;
  printarr(n,b);
  free(b);
}
```

C review: Dynamically allocated arrays

Example

```c
#include <stdlib.h>
#include <stdio.h>
void printarr(int n, int *a) {
  for (int i=0;i<n;i++)
    printf("%d ", a[i]);
  printf("\n");
}
int main(){
  int n=100;
  int *b=malloc(n*sizeof(*b));
  for (int i=0;i<n;i++)
    b[i]=i*i;
  printarr(n,b);
  free(b);
}
```

```
$ gcc -o dynarr dynarr.c -O2 -std=c99
$ ./dynarr
0 1 4 9 16 25 36 49 64 81 100 121 144
169 196 225 256 289 324 361 400 441
484 529 576 625 676 729 784 841 900
961 1024 1089 1156 1225 1296 1369 1444
1521 1600 1681 1764 1849 1936 2025
2116 2209 2304 2401 2500 2601 2704
2809 2916 3025 3136 3249 3364 3481
3600 3721 3844 3969 4096 4225 4356
4489 4624 4761 4900 5041 5184 5329
5476 5625 5776 5929 6084 6241 6400
6561 6724 6889 7056 7225 7396 7569
7744 7921 8100 8281 8464 8649 8836
9025 9216 9409 9604 9801
$ █
```

C review: Structs = collections of other variables

```
struct name {
  type1 name1;
  type2 name2;
  ...
};
```

C review: Structs = collections of other variables

```
struct name {
  type1 name1;
  type2 name2;
  ...
};
```

Example

```
#include <string.h>
#include <stdio.h>
struct Info {
  char name[100];
  unsigned int age;
};
int main() {
  struct Info my;
  my.age=38;
  strcpy(my.name,"Ramses");
  printf("%d %s\",my.age,my.name);
}
```

C review: Structs = collections of other variables

```
struct name {
  type1 name1;
  type2 name2;
  ...
};
```

Example

```c
#include <string.h>
#include <stdio.h>
struct Info {
  char name[100];
  unsigned int age;
};
int main() {
  struct Info my;
  my.age=38;
  strcpy(my.name,"Ramses");
  printf("%d %s\",my.age,my.name);
}
```

```
$ gcc -o info info.c -O2 -std=c99
$ ./info
Ramses 38
$ █
```

C review: Conditionals

```
if (condition) {
  statements
} else if (other condition) {
  statements
} else {
  statements
}
```

Example

C review: Conditionals

```
if (condition) {
  statements
} else if (other condition) {
  statements
} else {
  statements
}
```

Example

```
int main(){
  int n=20;
  int *b= malloc(n*sizeof(*b));
  if (b==0)
    return 1; //error
  else {
    for (int i=0;i<n;i++)
      b[i]=i*i;
    printarr(n,b);
    free(b);
  }
}
```

C review: Conditionals

```
if (condition) {
  statements
} else if (other condition) {
  statements
} else {
  statements
}
```

Example

```
int main(){
  int n=20;
  int *b= malloc(n*sizeof(*b));
  if (b==0)
    return 1; //error
  else {
    for (int i=0;i<n;i++)
      b[i]=i*i;
    printarr(n,b);
    free(b);
  }
}
```

```
$ gcc -o ifm ifm.c -O2 -std=c99
$ ./ifm
0 1 4 9 16 25 36 49 64 81 100 121
144 169 196 225 256 289 324 361
$ █
```

C review: Multidimensional arrays

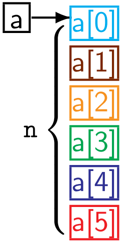
```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```

C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```

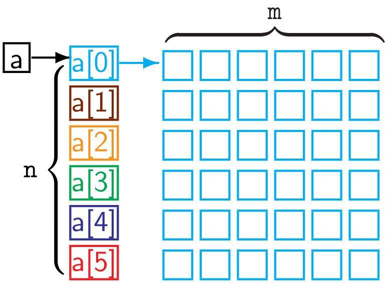
C review: Multidimensional arrays

```
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



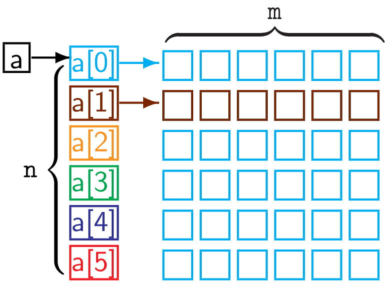
C review: Multidimensional arrays

```
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



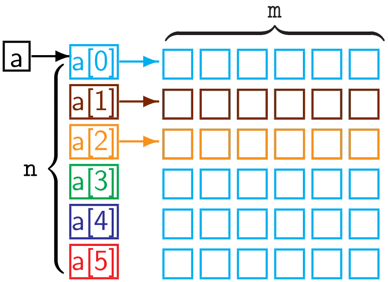
C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



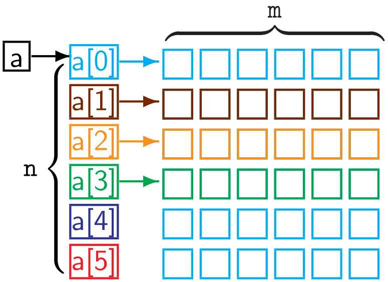
C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



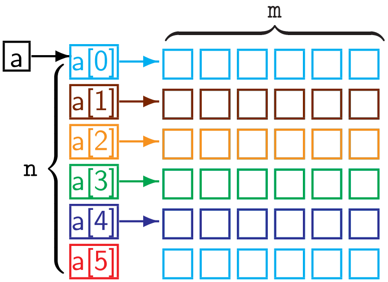
C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



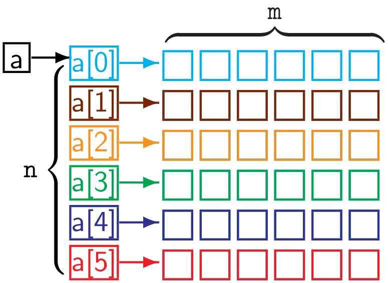
C review: Multidimensional arrays

```
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



C review: Multidimensional arrays

```c
#include <stdlib.h>
#include <assert.h>
float **matrix(long n,long m) {
  float **a=malloc(n*sizeof(*a));
  assert(a); // check if a not null
  a[0]=malloc(n*m*sizeof(**a));
  assert(a[0]); // check if a[0] not null
  for (long i=1; i<n; i++)
    a[i]=&a[0][i*m];
  return a;
}
void free_matrix(float **a) {
  free(a[0]);
  free(a);
}
void fill(long n,long m,float **a,float v){
  for (long i=0; i<n; i++)
    for (long j=0; j<m; j++)
      a[i][j]=v;
}
```



C review: Libraries

Usage

- Put an include line in the source code, e.g.

```
#include <stdio.h>
#include <omp.h>
#include "mpi.h"
```

- Include the libraries at link time using `-l[libname]`.
  Implicit for most standard libraries, with `mpicc` and `gcc -fopenmp`.

C review: Libraries

Usage

- Put an include line in the source code, e.g.

```
#include <stdio.h>
#include <omp.h>
#include "mpi.h"
```

- Include the libraries at link time using `-l[libname]`.
  Implicit for most standard libraries, with `mpicc` and `gcc -fopenmp`.

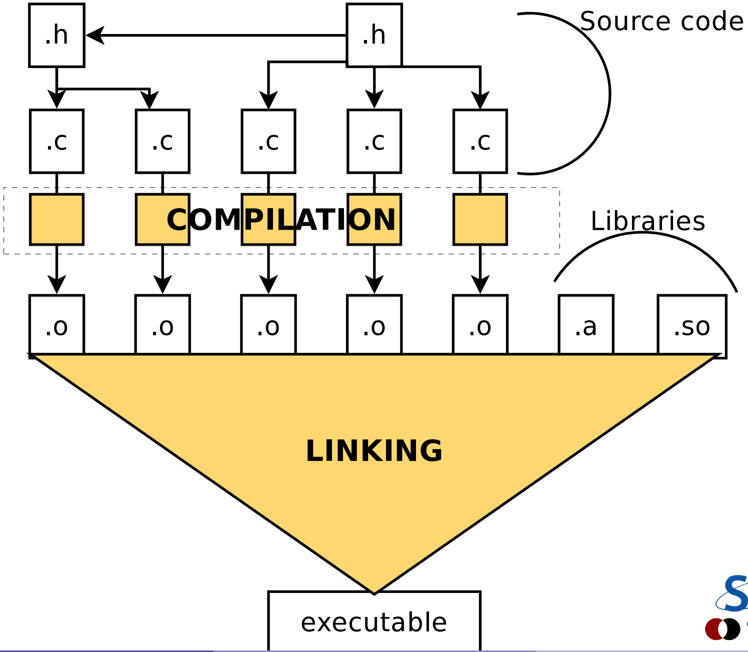
Common standard libraries

- stdio.h: input/output, e.g., `printf` and `fwrite`
- stdlib.h: memory, e.g. `malloc`
- string.h: strings, memory copies, e.g. `strcpy`
- math.h: special functions, e.g. `sqrt`.
  When using math, you need to link with `-lm`.

Compilation:

Building with make

Compilation workflow



Compiling with make

Single source file

```
# This file is called makefile
CC      = gcc
CFLAGS  = -std=c99 -O2
LDFLAGS = -lm
main:  main.c
   $(CC) $(CFLAGS) $(LDFLAGS) $^ -o $@
```

Compiling with make

Single source file

```
# This file is called makefile
CC      = gcc
CFLAGS  = -std=c99 -O2
LDFLAGS = -lm
main:  main.c
   $(CC) $(CFLAGS) $(LDFLAGS) $^ -o $@
```

Multiple source file application

```
CC      = gcc
CFLAGS  = -std=c99 -O2
LDFLAGS = -lm
main:  main.o mylib.o
   $(CC) $(LDFLAGS) $^ -o $@
main.o:  main.c mylib.h
mylib.o:  mylib.h mylib.c
clean:
   rm -f main.o mylib.o
```

Compiling with make

When typing `make` at command line:

- Checks if `main.c` or `mylib.c` or `mylib.h` were changed.
- If so, invokes corresponding rules for object files.
- Only compiles changed code files: faster recompilation.
- Parallel make:

```
$ make -j 3
```

Compiling with make

When typing `make` at command line:

- Checks if `main.c` or `mylib.c` or `mylib.h` were changed.
- If so, invokes corresponding rules for object files.
- Only compiles changed code files: faster recompilation.
- Parallel make:

```
$ make -j 3
```

Gotcha

- Make does not detect changes in compiler, or in system.
- But .o files are system/compiler dependent, so need to be recompiled.
- Always specify a "clean" rule in the makefile, so that moving from one system or compiler to another, you can do a fresh rebuild:

```
$ make clean
$ make
```