Part II

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.

Most basic C program:

```
int main() {
   return 0;
}
```

- main is first called function: must return an `int` .
- C expresses a lot with punctuation.

SciNet compute • calcul CANADA

C review: Language elements

Variables

Define a variable with

```
type name;
```

where *type* may be a

- built-in type:
  - floating point type:
    float, **double**, **long double**
  - integer type:
    short, [**unsigned**] **int**, [**unsigned**] **long int**
  - character or string of characters:
    char, **char***
- structure
- enumerated type
- union
- array
- pointer

C review: Language elements

Pointers

```
type *name;
```

Assignment:

```
int a,b;
int *ptr = &a;
a = 10;
b = *ptr;
```

Automatic arrays

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

Gotcha: limitations on automatic arrays

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

C review: Language elements

Dynamically allocated arrays

Defined as a pointer to memory:

```
type *name;
```

Allocated by a function call:

```
name = (type*)malloc(sizeof(type)*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 gived back.
- Can even resize memory.

Even better in C++

SciNet compute • calcul CANADA

C review: Language elements

Structures: collection of other variables.

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

Example

```
struct Info {
   char name[100];
   unsigned int age;
};
struct Info myinfo;
myinfo.age = 38;
strcpy(myinfo.name, "Ramses");
```

C review: Language elements

Enums

Used to define integer constants, typically increasing.

```
enum name {
   enumerator[=value], ...
};
```

By default, successive enumerators get successive integer values.

- In C, interconvertible with an int.
- Useful to reduce number of `#define`'s.

Unions

Put one variable on top of another; rarely used.

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

C review: Language elements

Typedefs

Used to give a name to an existing data type, or a compound data type.

```
typedef existingtype newtype;
```

Similar to *existingtype name*; but defines a type instead of a variable.

Example (a controversial way to get rid of the `struct` keyword)

```
typedef struct Info Info;
```

Then you can declare a `struct Info` simply by

```
Info myinfo;
```

This works become the name `Info` in "`struct Info`" does not live in the namespace of typenames.

SciNet compute • calcul CANADA

C review: Language elements

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 just functions with return type `void` and are called without assignment.

C review: Language elements

Conditionals

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

```
switch (integer-expression) {
   case integer:
      statements
      break;
   ...
   default:
      statements;
      break;
}
```

C review: Language elements

Loops

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

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

You can use **break** to exit the loop.

C review: Operators

C has many operators

```
()   []   ->   .
!         ++   --   (type)   -    *    &
*    /    %
+    -
<<   >>   <    <=   >        >=
==   !=
&    ^    |    &&   ||       ?:
=    +=   -=   *=   /=       %=   |=   &=
,
```

Gotcha: Bad precendence

Relying on operator precedence is error-prone and makes code harder to read and thus maintain (except for `+`, `-`, `*`, `/` and maybe `%`).

C review: Libraries

Usage

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

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

- Include the libraries at link time.
  (not needed for standard libraries)

Common standard libraries

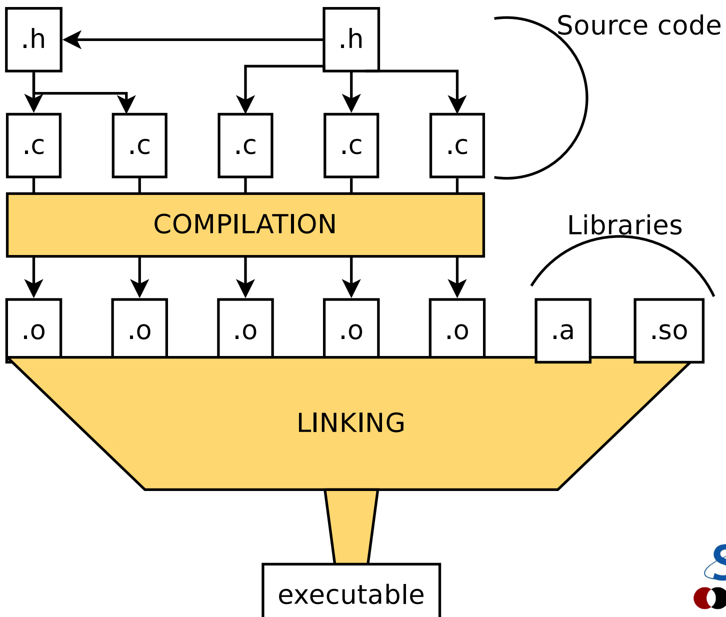
- stdio.h: input/output, e.g.,**`printf`** and **`fwrite`**
- stdlib.h: memory, e.g. **`malloc`**
- string.h: strings, memory copies, e.g. **`strcpy`**
- math.h: special function, e.g. **`sqrt`**

SciNet compute • calcul CANADA

Compilation:

Workflow

Compilation workflow



Compiling

Scientific computing = performance: Compile with optimization!

Compiling C from the command-line

If the source is in **main.c**, type

```
$ gcc main.c -O3 -o main
```

or

```
$ icc main.c -O3 -o main
```

Compiling C++ from the command-line

If the source is in **main.cpp**, type

```
$ g++ main.cpp -O3 -o main
```

or

```
$ icpc main.cpp -O3 -o main
```

Compilation:

Using make

Compiling with make

Single source file

```
# This file is called makefile
CC = gcc
CFLAGS = -O3
main:  main.c
    $(CC) $(CFLAGS) main.c -o main
```

Multiple source file application

```
CC = gcc
CFLAGS = -O3
main:  main.o mylib.o
    $(CC) main.o mylib.o -o main
main.o:  main.c mylib.h
mylib.o:  mylib.h mylib.c
clean:
    \rm 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.

Gotcha:

Make can only detect changes in the dependencies.
It does not detect changes in compiler, or in system.
But .o files are system/compiler dependent, so they should be recompiled.
So 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
```