Computational Astrophysics: The practical side

today:

01: Programming in C

S.R. Knollmann, UAM

last updated: 15.11.2010

Overview

- Overview
- The Language
  - technical terms
  - syntax description
- The Library
  - standard feature
- Everyday Usage
  - compiling
  - 'more than one file'
  - using libraries

Why C?

subjective, non-complete

- nearly universal availability
  - various implementations
  - a free implementation that works nearly everywhere: gcc
- extremely powerful
  - operating systems are generally written in C
  - can make use of specialized hardware relatively easy
- very mature
  - ISO standard (ISO/ECI 9899)
    http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
  - many people use it, so it is more likely that your code has a bug than the implementation
- very popular
  - TIOBE index October 2010: 1. Java 2. C 3. C++
  - lots of (free) documentation
  - lots of code to learn from
- related to other languages (most notably: C++, C#)
- allows you to do what you want (more or less)
  "The programmer is always right"-philosophy

"*All email clients suck. This one just sucks less.*."
– Michael Elkins

Why C?

subjective, non-complete

- nearly universal availability
  - various implementations
  - a free implementation that works nearly everywhere: gcc
- extremely powerful
  - operating systems are generally written in C
  - can make use of specialized hardware relatively easy
- very mature
  - ISO standard (ISO/ECI 9899)
    http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
  - many people use it, so it is more likely that your code has a bug than the implementation
- very popular
  - TIOBE index October 2010: 1. Java 2. C 3. C++
  - lots of (free) documentation
  - lots of code to learn from
- related to other languages (most notably: C++, C#)
- allows you to do what you want (more or less)
  "The programmer is always right"-philosophy

"*All email clients suck. This one just sucks less.*."
– Michael Elkins

Why C?

subjective, non-complete

- nearly universal availability
  - various implementations
  - a free implementation that works nearly everywhere: gcc
- extremely powerful
  - operating systems are generally written in C
  - can make use of specialized hardware relatively easy
- very mature
  - ISO standard (ISO/ECI 9899)
    http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
  - many people use it, so it is more likely that your code has a bug than the implementation
- very popular
  - TIOBE index October 2010: 1. Java 2. C 3. C++
  - lots of (free) documentation
  - lots of code to learn from
- related to other languages (most notably: C++, C#)
- allows you to do what you want (more or less)
  "The programmer is always right"-philosophy

"*All email clients suck. This one just sucks less..*"
– Michael Elkins

Why C?

subjective, non-complete

- nearly universal availability
  - various implementations
  - a free implementation that works nearly everywhere: gcc
- extremely powerful
  - operating systems are generally written in C
  - can make use of specialized hardware relatively easy
- very mature
  - ISO standard (ISO/ECI 9899)
    http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
  - many people use it, so it is more likely that your code has a bug than the implementation
- very popular
  - TIOBE index October 2010: 1. Java 2. C 3. C++
  - lots of (free) documentation
  - lots of code to learn from
- related to other languages (most notably: C++, C#)
- allows you to do what you want (more or less)
  "The programmer is always right"-philosophy

"All email clients suck. This one just sucks less.."
– Michael Elkins

Why C?

subjective, non-complete

- nearly universal availability
  - various implementations
  - a free implementation that works nearly everywhere: gcc
- extremely powerful
  - operating systems are generally written in C
  - can make use of specialized hardware relatively easy
- very mature
  - ISO standard (ISO/ECI 9899)
    http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
  - many people use it, so it is more likely that your code has a bug than the implementation
- very popular
  - TIOBE index October 2010: 1. Java 2. C 3. C++
  - lots of (free) documentation
  - lots of code to learn from
- related to other languages (most notably: C++, C#)
- allows you to do what you want (more or less)
  "The programmer is always right"-philosophy

"*All email clients suck. This one just sucks less..*"
– Michael Elkins

Why C?

subjective, non-complete

- nearly universal availability
  - various implementations
  - a free implementation that works nearly everywhere: gcc
- extremely powerful
  - operating systems are generally written in C
  - can make use of specialized hardware relatively easy
- very mature
  - ISO standard (ISO/ECI 9899)
    http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
  - many people use it, so it is more likely that your code has a bug than the implementation
- very popular
  - TIOBE index October 2010: 1. Java 2. C 3. C++
  - lots of (free) documentation
  - lots of code to learn from
- related to other languages (most notably: C++, C#)
- allows you to do what you want (more or less)
  "The programmer is always right"-philosophy

"All email clients suck. This one just sucks less.."
– Michael Elkins

Why C?

subjective, non-complete

- nearly universal availability
  - various implementations
  - a free implementation that works nearly everywhere: gcc
- extremely powerful
  - operating systems are generally written in C
  - can make use of specialized hardware relatively easy
- very mature
  - ISO standard (ISO/ECI 9899)
    http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf
  - many people use it, so it is more likely that your code has a bug than the implementation
- very popular
  - TIOBE index October 2010: 1. Java 2. C 3. C++
  - lots of (free) documentation
  - lots of code to learn from
- related to other languages (most notably: C++, C#)
- allows you to do what you want (more or less)
  "The programmer is always right"-philosophy

"*All email clients suck. This one just sucks less..*"
– Michael Elkins

History

(see also: http://en.wikipedia.org/wiki/C_programming_language#History)

1969-1973	initial development (by Dennis Ritchie at AT&T Bell Labs)
1973	Unix kernel is rewritten in C
1978	first edition of “The C Programming Language” known as K&R C (Brian Kernighan, Dennis Ritchie)
197?-198?	C (variants thereof) is implemented for a wide variety of mainframes, minicomputers and microcomputers (including IBM PC)
1983	American National Standards Institute (ANSI) forms the X3J11 committee
1989	ANSI X3.159-1989 "Programming Language C" aka ANSI-C, or C89
1990	The International Organization for Standardization (ISO) adopts ANSI-C as ISO/IEC 9899:1990, aka C90
1999	ISO/IEC 9899:1999, aka C99

example.c | example.c (equivalent)

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

double
power(double d, int n)
{
	int    i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int    n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

example.c | example.c (equivalent)

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

double power(double d, int n) {
int i;
double tmp = 1.0;
if(n>0)
{for(i=0;i<n;i++)
tmp = tmp * d;
}else{
double dInv = 1. / d;
for(i=0;i<-n;i++)
tmp=tmp*dInv;
}
return tmp;}

int main(int argc, char **argv) {
double d;
int n;
d = atof(argv[1]);
n = atoi(argv[2]);
printf("%f^%i = %f \n",d,n,power(d, n)); return 0;}
```

example.c | example.c (equivalent)

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

double power(double d, int n) {
int i;
double tmp = 1.0;
if(n>0)
{for(i=0;i<n;i++)
tmp = tmp * d;
}else{
double dInv = 1. / d;
for(i=0;i<-n;i++)
tmp=tmp*dInv;
}
return tmp;}

int main(int argc, char **argv) {
double d;
int n;
d = atof(argv[1]);
n = atoi(argv[2]);
printf("%f^%i = %f \n",d,n,power(d, n)); return 0;}
```

example.c

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

double
power(double d, int n)
{
	int	i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

example.c

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

double
power(double d, int n)
{
	int	i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

Functions

example.c

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

double
power(double d, int n)
{
	int	i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

Blocks

example.c

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

double
power(double d, int n)
{
	int	i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

Loops

Flow-Control

example.c

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

double
power(double d, int n)
{
	int	i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

example.c

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

double
power(double d, int n)
{
	int	i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

Stack Changes

example.c

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

double
power(double d, int n)
{
	int	i;
	double	tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double	d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

Types

example.c

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

double
power(double d, int n)
{
	int	i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

Variables

example.c

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

double
power(double d, int n)
{
	int    i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int    n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

Variables

Note:

First declare then use

example.c

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

double
power(double d, int n)
{
	int	i;
	double tmp = 1.0;

	if (n > 0) {
		for (i = 0; i < n; i++)
			tmp = tmp * d;
	} else {
		double dInv = 1. / d;
		for (i = 0; i < -n; i++)
			tmp = tmp * dInv;
	}

	return tmp;
}

int
main(int argc, char **argv)
{
	double d;
	int	n;

	d = atof(argv[1]);
	n = atoi(argv[2]);
	printf("%f^%i = %f \n", d, n, power(d, n));

	return 0;
}
```

Preprocessor

Allowed characters

Two sets:

- *source character set*
  what the code is written in
- *execution character set*
  what gets interpreted by the execution environment

Basic (source and execution):

- 26 uppercase and 26 lowercase Latin characters, 10 digits, 29 graphical characters:
  `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
  `a b c d e f g h i j k l m n o p q r s t u v w x y z`
  `0 1 2 3 4 5 6 7 8 9 0`
  `! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ { | } ~`
- Space character, control characters representing horizontal tab, vertical tab, and form feed
- In source set: A way to indicate the end of a line
- In execution set: control characters for alert, backspace, carriage return, and new line

In string literals or character constants (in the source file):

- Execution characters are expressed by their corresponding source character, or by an escape sequence.
- A byte with all bits set to 0, the *null character*, is used to terminate strings.

Allowed characters

Two sets:

- *source character set*
  what the code is written in
- *execution character set*
  what gets interpreted by the execution environment

Basic (source and execution):

- 26 uppercase and 26 lowercase Latin characters, 10 digits, 29 graphical characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! “ # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ { | } ~
```

- Space character, control characters representing horizontal tab, vertical tab, and form feed
- In source set: A way to indicate the end of a line
- In execution set: control characters for alert, backspace, carriage return, and new line

In string literals or character constants (in the source file):

- Execution characters are expressed by their corresponding source character, or by an escape sequence.
- A byte with all bits set to 0, the *null character*, is used to terminate strings.

Allowed characters

Two sets:

- *source character set*
  what the code is written in
- *execution character set*
  what gets interpreted by the execution environment

Basic (source and execution):

- 26 uppercase and 26 lowercase Latin characters, 10 digits, 29 graphical characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ { | } ~
```

- Space character, control characters representing horizontal tab, vertical tab, and form feed
- In source set: A way to indicate the end of a line
- In execution set: control characters for alert, backspace, carriage return, and new line

In string literals or character constants (in the source file):

- Execution characters are expressed by their corresponding source character, or by an escape sequence.
- A byte with all bits set to 0, the *null character*, is used to terminate strings.

Allowed characters

Two sets:

- *source character set*
  what the code is written in
- *execution character set*
  what gets interpreted by the execution environment

Basic (source and execution):

- 26 uppercase and 26 lowercase Latin characters, 10 digits, 29 graphical characters:
  ```
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  0 1 2 3 4 5 6 7 8 9
  ! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ { | } ~
  ```
- Space character, control characters representing horizontal tab, vertical tab, and form feed
- In source set: A way to indicate the end of a line
- In execution set: control characters for alert, backspace, carriage return, and new line

In string literals or character constants (in the source file):

- Execution characters are expressed by their corresponding source character, or by an escape sequence.
- A byte with all bits set to 0, the *null character*, is used to terminate strings.

Escape sequences:

\a	Alert
\b	Backspace (move cursor one position to the left)
\f	Formfeed (move to the next page)
\n	Newline
\r	Carriage return (move cursor to beginning of line)
\t	Horizontal tab
\v	Vertical tab
\"	Print "
\?	Print ?
\'	Print '
****	Print \
\0	Null character

Allowed characters

```
!!!WARNING!!!

Those are ASCII characters.  When copy and pasting from a
website, typographical characters not equal to the ASCII
characters can make their way in your source.  This will
produce funny errors.

E.g.: — – vs. -
      ” “ vs. “
      ’ ‘ vs. '
```

Two sets:

- *source character set*
  what the code is written in
- *execution character set*
  what gets interpreted by the execution environment

Basic (source and execution):

- 26 uppercase and 26 lowercase Latin characters, 10 digits, 29 graphical characters:
  `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
  `a b c d e f g h i j k l m n o p q r s t u v w x y z`
  `0 1 2 3 4 5 6 7 8 9`
  `! “ # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ { | } ~`
- Space character, control characters representing horizontal tab, vertical tab, and form feed
- In source set: A way to indicate the end of a line
- In execution set: control characters for alert, backspace, carriage return, and new line

In string literals or character constants (in the source file):

- Execution characters are expressed by their corresponding source character, or by an escape sequence.
- A byte with all bits set to 0, the *null character*, is used to terminate strings.

Keywords

List of keywords (C99):

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

Identifiers

Can denote:

- an object
- a function
- a tag or a member of a structure, union, or enumeration
- a typedef name
- a label name
- a macro name
- a macro parameter

Have:

- *scope*
  the region in which the identifier is known
- *linkage*
  defines whether the same name in a different scope refers to the same identifier
- *name space*
  can allow to have the same identifier visible at a given time (though referring to different things)

Identifiers

Valid identifiers:

- Can contain:
  `_ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`
  `a b c d e f g h i j k l m n o p q r s t u v w x y z`
  `0 1 2 3 4 5 6 7 8 9`
- must not start with a digit
- are case-sensitive
- identifiers starting with _ should be avoided (often used internally by the implementation)
- identifiers must be different from keywords

```
valid:
hello, hElLO_231, bla, foobar, FOOBAR, f1, ...

valid, but avoid:
_my, _00231, _hdas32, ...

not valid:
0hello, 1HELLO, for, while, _Bool, ...
```

Identifiers: Scope

Possible scopes:

- *function*
  only labels
- *file*
  if declarator appears outside of any block or list of parameters
  terminates at the end of the translation unit (approximately: end of source file it is in)
- *block*
  if declarator appears inside a block or list of parameter declarations in a function definition
  terminates at the end of the associated block
- *function prototype*
  if declarators appears inside a list of parameters in a function prototype (not its definition)
  terminate at the end of the function declarator

'Shadowing'

- scopes can overlap (e.g. same identifier in nested blocks)
- within the inner scope, the identifier refers to the entity declared in the inner scope:
  the entity of the outer scope is *hidden*, or *shadowed*.
- within the outer scope, the identifier refers to the entity declared in the outer scope

Identifiers: Scope

Possible scopes:

- *function*
  only labels
- *file*
  if declarator appears outside of any block or list of parameters
  terminates at the end of the translation unit (approximately: end of source file it is in)
- *block*
  if declarator appears inside a block or list of parameter declarations in a function definition
  terminates at the end of the associated block
- *function prototype*
  if declarators appears inside a list of parameters in a function prototype (not its definition)
  terminate at the end of the function declarator

'Shadowing'

- scopes can overlap (e.g. same identifier in nested blocks)
- within the inner scope, the identifier refers to the entity declared in the inner scope:
  the entity of the outer scope is *hidden*, or *shadowed*.
- within the outer scope, the identifier refers to the entity declared in the outer scope

Identifiers: Linkage

> "An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*."
>
> ISO/IEC 9899:TC2

External

- in the entire program (constituted by a set of translation units and libraries) identifiers with external linkage refer to the same object or function
- indicated by storage class **extern**
- if a function declaration has no explicit storage declaration, it is **extern**
- if a declaration for an object has file scope and no explicit storage declaration

Internal

- within a translation unit, an identifier of internal linkage denotes the same object or function
- indicated by storage class **static**
- if the storage class of a file scope identifier of an object or a function is **static**

None

- identifiers with no linkage denote a unique entity
- identifiers to be anything but a function or an object
- an identifier declared to be a function parameter
- block scope identifier for an object without the storage class **extern**

Identifiers: Linkage

> "An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*."
>
> ISO/IEC 9899:TC2

External

- in the entire program (constituted by a set of translation units and libraries) identifiers with external linkage refer to the same object or function
- indicated by storage class **extern**
- if a function declaration has no explicit storage declaration, it is **extern**
- if a declaration for an object has file scope and no explicit storage declaration

Internal

- within a translation unit, an identifier of internal linkage denotes the same object or function
- indicated by storage class **static**
- if the storage class of a file scope identifier of an object or a function is **static**

None

- identifiers with no linkage denote a unique entity
- identifiers to be anything but a function or an object
- an identifier declared to be a function parameter
- block scope identifier for an object without the storage class **extern**

Identifiers: Linkage

> "An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*."
>
> ISO/IEC 9899:TC2

External

- in the entire program (constituted by a set of translation units and libraries) identifiers with external linkage refer to the same object or function
- indicated by storage class **extern**
- if a function declaration has no explicit storage declaration, it is **extern**
- if a declaration for an object has file scope and no explicit storage declaration

Internal

- within a translation unit, an identifier of internal linkage denotes the same object or function
- indicated by storage class **static**
- if the storage class of a file scope identifier of an object or a function is **static**

None

- identifiers with no linkage denote a unique entity
- identifiers to be anything but a function or an object
- an identifier declared to be a function parameter
- block scope identifier for an object without the storage class **extern**

Identifiers: Linkage

> "An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*."
>
> ISO/IEC 9899:TC2

External

- in the entire program (constituted by a set of translation units and libraries) identifiers with external linkage refer to the same object or function
- indicated by storage class **extern**
- if a function declaration has no explicit storage declaration, it is **extern**
- if a declaration for an object has file scope and no explicit storage declaration

Internal

- within a translation unit, an identifier of internal linkage denotes the same object or function
- indicated by storage class **static**
- if the storage class of a file scope identifier of an object or a function is **static**

None

- identifiers with no linkage denote a unique entity
- identifiers to be anything but a function or an object
- an identifier declared to be a function parameter
- block scope identifier for an object without the storage class **extern**

Identifiers: Name Space

Possible Names Spaces

- *label name*
  for **goto** or **switch**
  disambiguated by: usage and declaration
- *tags*
  for structures, unions, enumerations
  disambiguated by: keywords **struct**, **union**, **enum**
- *members*
  each structure and union has a name space for its members
  disambiguated by: the access method (**.** or **->** operator)
- *ordinary identifiers*
  all other identifiers

Identifiers: Name Space

Possible Names Spaces

- *label name*
  for **goto** or **switch**
  disambiguated by: usage and declaration
- *tags*
  for structures, unions, enumerations
  disambiguated by: keywords **struct**, **union**, **enum**
- *members*
  each structure and union has a name space for its members
  disambiguated by: the access method (**.** or **->** operator)
- *ordinary identifiers*
  all other identifiers

```c
#include <stdio.h>

struct hello {
    int hello;
};

int
main(void)
{
    int           hello = 1;
    struct hello helloStruct;

    helloStruct.hello = hello;

    printf("%i\n%i\n",
           hello,
           helloStruct.hello);

hello:
    return 0;
}
```

Identifiers: Lifetime of objects

Lifetime

Duration during which storage is reserved for an object. During that time it will

- have a constant address
- retain its last-stored value

Note:

- using objects outside their lifetime is undefined
- pointers to objects outside their lifetime become undefined

Possible lifetimes (storage durations)

- static
  objects with external or internal linkage, or with the storage-class static
  will be initialized once before program startup and is available during the whole runtime
- automatic
  objects with no linkage and without storage-class static
  come into existence when block they are associated with is entered
  lifetime ends, when their associated block is left in any way (function calls are superseding the block, not leaving it)
- allocated
  Programmer has to deal with memory allocation (library functionality)

Identifiers: Lifetime of objects

Lifetime

Duration during which storage is reserved for an object. During that time it will

- have a constant address
- retain its last-stored value

Note:

- using objects outside their lifetime is undefined
- pointers to objects outside their lifetime become undefined

Possible lifetimes (storage durations)

- static
  objects with external or internal linkage, or with the storage-class static
  will be initialized once before program startup and is available during the whole runtime
- automatic
  objects with no linkage and without storage-class static
  come into existence when block they are associated with is entered
  lifetime ends, when their associated block is left in any way (function calls are superseding the block, not leaving it)
- allocated
  Programmer has to deal with memory allocation (library functionality)

Types

Object Types

- types describing objects

Derived Types

- constructed from basic types

Incomplete Types

- types that describe the objects, but lack information to calculate their sizes

Types: Object Types

Integer types:

- `_Bool`
  large enough to store 0 and 1
- `char`
  large enough to store any member of the basic execution set (they will all have positive values)
- standard signed integer types:
  `signed char`, `short int`, `int`, `long int`, `long long int`
  Beware: sizes can vary between architectures!
- standard unsigned integer types:
  `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`

Real floating types:

- `float`, `double`, `long double`

Complex floating types:

- `float _Complex`, `double _Complex`, `long double _Complex`

More:

- Integer and floating types are called arithmetic types (two domains: real and complex)
- `void`: empty set of values (incomplete type)

Types: Object Types

```
Note:
Instead of _Bool    : bool
                      requires stdbool.h
Instead of _Complex: complex
                      requires complex.h
```

Integer types:

- `_Bool`
  large enough to store 0 and 1
- `char`
  large enough to store any member of the basic execution set (they will all have positive values)
- standard signed integer types:
  `signed char`, `short int`, `int`, `long int`, `long long int`
  Beware: sizes can vary between architectures!
- standard unsigned integer types:
  `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`, `unsigned long long int`

Real floating types:

- `float`, `double`, `long double`

Complex floating types:

- `float _Complex`, `double _Complex`, `long double _Complex`

More:

- Integer and floating types are called arithmetic types (two domains: real and complex)
- `void`: empty set of values (incomplete type)

Types: Derived Types

Arrays:

- contiguously allocated nonempty set of objects of a given type

```
double arr[128];
```

```
myType_t arr[128];
```

Structure:

- sequentially allocated nonempty collections of objects (may be of different types)

Unions:

- like structure but overlapping

Types: Derived Types

Arrays:

- contiguously allocated nonempty set of objects of a given type

```
double arr[128];
```

```
myType_t arr[128];
```

Structure:

- sequentially allocated nonempty collections of objects (may be of different types)

```
struct tag {
    int    id;
    char   *name;
    double x[128];
    double y[128];
} myStruct;
```

```
myStruct.id = 1;
myStruct.name = “Funny Name”;
for (int i = 0; i < 128; i++) {
    myStruct.x[i] = (double)(i + 1);
    myStruct.y[i] = log(myStruct.x[i]);
}
```

Unions:

- like structure but overlapping

Types: Derived Types

Arrays:

- contiguously allocated nonempty set of objects of a given type

```
double arr[128];
```

```
myType_t arr[128];
```

Structure:

- sequentially allocated nonempty collections of objects (may be of different types)

```
struct tag {
    int    id;
    char   *name;
    double x[128];
    double y[128];
} myStruct;
```

```
myStruct.id = 1;
myStruct.name = "Funny Name";
for (int i = 0; i < 128; i++) {
    myStruct.x[i] = (double)(i + 1);
    myStruct.y[i] = log(myStruct.x[i]);
}
```

Unions:

- like structure but overlapping

```
union tag {
    char    c;
    int     i;
    double  d;
    float   f;
} myUnion;
```

Types: Derived Types

Pointer:

- may be derived from a function type, an object type, or an incomplete type, which is called the *referenced type*
- value (i.e. memory address) is a reference to an entity of the referenced type

Simple pointers

```
struct tag {
    int    id;
};

// More things

{
    struct tag myStruct, *myStructPtr;
    int        *idPtr;

    myStructPtr = &myStruct;

    myStructPtr->id = 1;
    idPtr           = &(myStruct.id);
    assert(*idPtr == myStructPtr->id);
}
```

Types: Derived Types

Pointer:

- may be derived from a function type, an object type, or an incomplete type, which is called the *referenced type*
- value (i.e. memory address) is a reference to an entity of the referenced type

Simple pointers

```
struct tag {
    int    id;
};

// More things

{
    struct tag myStruct, *myStructPtr;
    int        *idPtr;

    myStructPtr = &myStruct;

    myStructPtr->id = 1;
    idPtr           = &(myStruct.id);
    assert(*idPtr == myStructPtr->id);
}
```

Function Pointers

```
extern int
compareDouble(const void *p1, const void *p2)
{
    if ( *((double *)p1) < *((double *)p2) )
        return -1;

    if ( *((double *)p1) > *((double *)p2) )
        return 1;

    return 0;
}

// More things

{
    double arr[128];

    // More things

    qsort(arr, 128, sizeof(double),
          &compareDouble);
}
```

Types: Derived Types

Functions:

- characterized by its return type and the number and types of its parameters

```
static int
myFunc(int d, double a, char *f);

static void
myFunc(int d, double a, char *f)
{
     return -4;
}
```

```
extern void
myFunc(void);

extern void
myFunc(void)
{
     // Do something, but don't return
}
```

```
myType                  myType
myFunc(myType s);       myFunc(myType);

myType
myFunc(myType s)
{
     return s;
}
```

Types: Derived Types

Functions:

- call-by-value!
  but passing a reference is possible

```c
void
myFunc(int a)
{
    a = 5;
}

int
main(void)
{
    int a = 1;

    // This will print 'a = 1'
    printf("a = %i\n", a);
    myFunc(a);
    // This will also print 'a = 1'
    printf("a = %i\n", a);

    return 0;
}
```

Types: Derived Types

Functions:

- call-by-value!
  but passing a reference is possible

```c
void
myFunc(int a)
{
    a = 5;
}

int
main(void)
{
    int a = 1;

    // This will print 'a = 1'
    printf("a = %i\n", a);
    myFunc(a);
    // This will also print 'a = 1'
    printf("a = %i\n", a);

    return 0;
}
```

```c
void
myFunc(int *a)
{
    *a = 5;
}

int
main(void)
{
    int a = 1;

    // This will print 'a = 1'
    printf("a = %i\n", a);
    myFunc(&a);
    // This will now print 'a = 5'
    printf("a = %i\n", a);

    return 0;
}
```

Types: Derived Types

Functions: Main

Starting point of the execution

Two allowed signatures

```
int
main(void);
```

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

If second form, then:
argc: Number of command line arguments
argv: Array of Strings holding the arguments

```
./myProg Haha 4.2332

argc = 3
argv[0] = "./myProg"
argv[1] = "Haha"
argv[2] = "4.2332"
argv[3] = NULL
```

Types: Derived Types

Functions: Main

Starting point of the execution

Two allowed signatures

```
int
main(void);
```

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

If second form, then:
argc: Number of command line arguments
argv: Array of Strings holding the arguments

```
./myProg Haha 4.2332

argc = 3
argv[0] = “./myProg”
argv[1] = “Haha”
argv[2] = “4.2332”
argv[3] = NULL
```

Expressions

- Primary Expressions
- Postfix operators
- Unary operators
- Cast operators
- Multiplicative operators
- Additive operators
- Relations
- Logical operators
- Conditional operator
- Assignment operator
- Bitwise operators
- Comma operator

Expressions: Primary expressions

- identifiers
  - if it has been declared as an object (*lvalue*)
    Note: undeclared identifiers are syntax errors
  - if it is a function (*function designator*)
- a constant
- string literal
- a parenthesized expression

Expressions: Postfix operators

- array subscripting
- function calls
- structure and union members
- increment and decrement
- compound literals

Expressions: Postfix operators

- **array subscripting**
- function calls
- structure and union members
- increment and decrement
- compound literals

```
E1[E2] is equivalent to *(E1 + E2), e.g.

double arr[128];
arr[0]  == *arr;
arr[45] == *(arr + 45);


double arr[5][5];
arr[2][3] == *(arr + (2 * 5) + 3);
```

Expressions: Postfix operators

- array subscripting
- **function calls**
- structure and union members
- increment and decrement
- compound literals

```
int
f(int a, double b);


int foo = 1;
float bar = -1.04;

f(foo, bar); // bar is promoted to double
```

Expressions: Postfix operators

- array subscripting
- function calls
- **structure and union members**
- increment and decrement
- compound literals

```
struct bla {
    int a;
    double b;
};

struct bla s, *sp;

sp = &s;

sp->a = 1;
s.b = 1.0;
```

Expressions: Postfix operators

- array subscripting
- function calls
- structure and union members
- **increment and decrement**
- compound literals

```c
int a = 1;

a++; // Identical to a = a + 1;
a--; // Identical to a = a – 1;


double arr[128];
double *dp = arr;

for (int i = 0; i < 128; i++) {
    *dp = 1.0;
    dp++;
    // Identical to arr[i] = 1.0
}
```

Expressions: Postfix operators

- array subscripting
- function calls
- structure and union members
- increment and decrement
- **compound literals**

```
int a[5] = {0, 1, 2, 3, 4};


drawline((struct point){.x = 1, .y = 4},
         (struct point){.x = 3, .y = 3});
```

Expressions: Unary operators

- **Prefix in- and decrement**
- Address and indirection
- Unary arithmetic operations
- sizeof operator

```
int a = 1;

--a;  // Equivalent to (a = a – 1);
++a;  // Equivalent to (a = a + 1);


Note:
double b[3] = {0., 0., 0.};
int i = 0;

b[++i] = 1.0; // is b[1] = 1.0;
b[i++] = 1.0; // is b[1] = 1.0;
b[i]   = 1.0; // is b[2] = 1.0;
```

Expressions: Unary operators

- Prefix in- and decrement
- **Address and indirection**
- Unary arithmetic operations
- sizeof operator

```
int a, *ap;

a = 1;
ap = &a;  // & is the address operator
a = *ap;  // * is the indirection
```

Expressions: Unary operators

- Prefix in- and decrement
- Address and indirection
- **Unary arithmetic operations**
- sizeof operator

```
!OP  → Logical negation !OP: (0 == OP)
+OP  → OP
-OP  → -OP
~OP  → bitwise complement (OP must be integer)

double a = f();
if (!isfinite(a))
```

Expressions: Unary operators

- Prefix in- and decrement
- Address and indirection
- Unary arithmetic operations
- **sizeof operator**

```c
double a = f();
size_t size;

size = sizeof(a); // storage size in bytes of a

size = sizeof(double);  // storage size in
                        // bytes for doubles

size = sizeof(struct tag);  // storage size in
                            // bytes for the
                            // tag structure

sizeof(char) == 1
```

Expressions: Cast operators

- Explicitly converts types

```
long int a = 990;
int      b;

b = (int)a;


void  *p;
double a;

p = (void *)(&a);
```

Expressions: Multiplicative operators

```
int a = 4;
int b = 3;

a * b  // 12
a / b  // 1
a / ((double)b) // 1.333333...
a % b  // 3
```

Expressions: Additive operators

Expressions: Multiplicative operators

```
int a = 4;
int b = 3;

a * b  // 12
a / b  // 1
a / ((double)b) // 1.333333...
a % b  // 3
```

Expressions: Additive operators

```
unsigned int a = 1;
unsigned int b = 2;

a + b  // 2
a - b  // 2^32 - 1
```

Expressions: Relations

```
int a = 4;
int b = 4;

a < b   // 1, i.e false
a > b   // 0, i.e. true
a <= b  // 0, i.e. true
a >= b  // 0, i.e. true
```

Expressions: Logical Operators

Expressions: Conditional Operator

Expressions: Relations

```
int a = 4;
int b = 4;

a < b   // 1, i.e false
a > b   // 0, i.e. true
a <= b  // 0, i.e. true
a >= b  // 0, i.e. true
```

Expressions: Logical Operators

```
==  Logical equal, e.g. a == b
!=  Logical not equal, e.g. a != b
&&  Logical AND, e.g. (a < 1) && (b > 2)
||  Logical OR, e.g. (a < 1) || (a > 1)
```

Expressions: Conditional Operator

Expressions: Relations

```
int a = 4;
int b = 4;

a < b   // 1, i.e false
a > b   // 0, i.e. true
a <= b  // 0, i.e. true
a >= b  // 0, i.e. true
```

Expressions: Logical Operators

```
==  Logical equal, e.g. a == b
!=  Logical not equal, e.g. a != b
&&  Logical AND, e.g. (a < 1) && (b > 2)
||  Logical OR, e.g. (a < 1) || (a > 1)
```

Expressions: Conditional Operator

```
int a = 4;
int b;

b = (a > 3) ? 34 : 12;
```

Expressions: Bitwise operators

```
a & b // bitwise AND
a | b // bitwise INCLUSIVE OR
a ^ b // bitwise EXCLUSIVE OR
a << b // left shift bits of a by b
a >> b // right shift bits of a by b
```

Expressions: Assignment operators

Expressions: Comma operator

Expressions: Bitwise operators

```
a & b // bitwise AND
a | b // bitwise INCLUSIVE OR
a ^ b // bitwise EXCLUSIVE OR
a << b // left shift bits of a by b
a >> b // right shift bits of a by b
```

Expressions: Assignment operators

```
a += b;       a <<= b;
a -= b;       a >>= b;
a *= b;       a &= b;
a /= b;       a |= b;
a %= b;       a ^= b;
```

Expressions: Comma operator

Expressions: Bitwise operators

```
a & b // bitwise AND
a | b // bitwise INCLUSIVE OR
a ^ b // bitwise EXCLUSIVE OR
a << b // left shift bits of a by b
a >> b // right shift bits of a by b
```

Expressions: Assignment operators

```
a += b;       a <<= b;
a -= b;       a >>= b;
a *= b;       a &= b;
a /= b;       a |= b;
a %= b;       a ^= b;
```

Expressions: Comma operator

```
int a;

a = a = 1, a + 3; // a = 4
```

Statements

A statement specifies an action to be performed.

- *labeled* statement
- *compound* statement
- *expression* and *null* statement
- *selection* statement (`if`, `switch`)
- *iteration* statement (`for`, `do`, `while`)
- *jump* statement (`goto`, `continue`, `break`, `return`)

Statements: Labeled

- provide a way to jump to specific points
- only to be used in selection statements
  `goto` is evil

```
labeled statement:
    identifier : statement
    case constant expression : statement
    default : statement
```

Statements: Compound

- compound statements are blocks

Statements: Expression, Null

- an expression statement is written as
  expression ;
- the *expression* is optional
  if omitted: null statement

Statements: Labeled

- provide a way to jump to specific points
- only to be used in selection statements
  `goto` is evil

```
labeled statement:
     identifier : statement
     case constant expression : statement
     default : statement
```

Statements: Compound

- compound statements are blocks

```
compound statement:
     { block-item-list_opt }
block-item-list:
     block-item
     block-item-list block-item
block-item:
     declaration
     statement
```

Statements: Expression, Null

- an expression statement is written as
  expression ;
- the *expression* is optional
  if omitted: null statement

Statements: Labeled

- provide a way to jump to specific points
- only to be used in selection statements
  `goto` is evil

```
labeled statement:
     identifier : statement
     case constant expression : statement
     default : statement
```

Statements: Compound

- compound statements are blocks

```
compound statement:
     { block-item-list_opt }
block-item-list:
     block-item
     block-item-list block-item
block-item:
     declaration
     statement
```

Statements: Expression, Null

- an expression statement is written as *expression* ;
- the *expression* is optional
  if omitted: null statement

```
expression statement:
     expression_opt ;
```

Statements: Selection

- selects among a set of statements depending on the value of the controlling expression
- is a block
- Note: for `if` (and `if/else`) selections, the first statement is executed iff the expression compares unequal to 0
- Note: in `switch` selections, the program flow jumps to the corresponding case and continues from there (possibly entering other cases)

```
selection statement:
    if ( expression ) statement
    if ( expression ) statement else statement
    switch ( expression ) statement
```

Statements: Selection

- selects among a set of statements depending on the value of the controlling expression
- is a block
- Note: for `if` (and `if/else`) selections, the first statement is executed iff the expression compares unequal to 0
- Note: in `switch` selections, the program flow jumps to the corresponding case and continues from there (possibly entering other cases)

```
if (a != 0)  // if (a)
     foo();


if (a > 0)
     foo();
else
     bar();


if ( (a > 0) != 0 ) {
     foo();
} else {
     bar();
}
```

Statements: Selection

- selects among a set of statements depending on the value of the controlling expression
- is a block
- Note: for `if` (and `if/else`) selections, the first statement is executed iff the expression compares unequal to 0
- Note: in `switch` selections, the program flow jumps to the corresponding case and continues from there (possibly entering other cases)

```
if (a != 0)  // if (a)
     foo();


if (a > 0)
     foo();
else
     bar();


if ( (a > 0) != 0 ) {
     foo();
} else {
     bar();
}
```

```
switch (a) {
case 0:
     foo();
case 1:
     bar();
default:
     ;
}
```

Statements: Selection

- selects among a set of statements depending on the value of the controlling expression
- is a block
- Note: for `if` (and `if/else`) selections, the first statement is executed iff the expression compares unequal to 0
- Note: in `switch` selections, the program flow jumps to the corresponding case and continues from there (possibly entering other cases)

```
if (a != 0)  // if (a)
     foo();


if (a > 0)
     foo();
else
     bar();


if ( (a > 0) != 0 ) {
     foo();
} else {
     bar();
}
```

```
switch (a) {
case 0:
     foo();
case 1:
     bar();
default:
     ;
}
```

```
switch (type) {
case TYPE_RED:
     red_foo();
     break;
case TYPE_YELLOW:
     yellow_foo();
     break;
case TYPE_GREEN:
     green_foo();
     break;
case TYPE_BLUE:
     blue_foo();
     break;
default:
     bar();
}
```

Statements: Iteration

- causes a statement (call the loop body) to be executed until the controlling expression compares equal to 0 (i.e. 'is false').
- Note: the controlling expression is evaluate before (after) the loop body for `while` (`do`) loops.
- Note: the second expression in the for loop is the controlling expression and if omitted is replaced with a non zero constant ('loop forever')
- Note: for `for` iterations, the declaration part can only declare variables of storage class **auto** or **register**

```
iteration statement:
    while ( expression ) statement
    do statement while ( expression ) ;
    for ( expression_opt ; expression_opt ; expression_opt ) statement
```

Statements: Iteration

- causes a statement (call the loop body) to be executed until the controlling expression compares equal to 0 (i.e. 'is false').
- Note: the controlling expression is evaluate before (after) the loop body for `while` (`do`) loops.
- Note: the second expression in the for loop is the controlling expression and if omitted is replaced with a non zero constant ('loop forever')
- Note: for `for` iterations, the declaration part can only declare variables of storage class **auto** or **register**

```
do
    a = foo()
while (a != 5);


do {
    a = foo();
    bar();
} while (a != 5);
```

Statements: Iteration

- causes a statement (call the loop body) to be executed until the controlling expression compares equal to 0 (i.e. 'is false').
- Note: the controlling expression is evaluate before (after) the loop body for `while` (`do`) loops.
- Note: the second expression in the for loop is the controlling expression and if omitted is replaced with a non zero constant ('loop forever')
- Note: for `for` iterations, the declaration part can only declare variables of storage class **auto** or **register**

```
do
     a = foo()
while (a != 5);


do {
     a = foo();
     bar();
} while (a != 5);
```

```
while (a != 5)
     a = foo();

while (a != 5) {
     a = foo();
     bar();
}

while (*s++ != '\0')
     ;
```

Statements: Iteration

- causes a statement (call the loop body) to be executed until the controlling expression compares equal to 0 (i.e. 'is false').
- Note: the controlling expression is evaluate before (after) the loop body for `while` (`do`) loops.
- Note: the second expression in the for loop is the controlling expression and if omitted is replaced with a non zero constant ('loop forever')
- Note: for `for` iterations, the declaration part can only declare variables of storage class **auto** or **register**

```
do
     a = foo()
while (a != 5);


do {
     a = foo();
     bar();
} while (a != 5);
```

```
while (a != 5)
     a = foo();

while (a != 5) {
     a = foo();
     bar();
}

while (*s++ != '\0')
     ;
```

```
for (i=5; i>=0; i--)
     foo();

for (; i>0; i--) {
     foo();
     bar();
}

for (int j=0;
     j<(1<<30);
     j++) {
     foo();
     bar();
}
```

Statements: Jump

- will cause the program flow to jump to the specified position
- **goto** is evil!
- Note: Not to be confused with the library jump functionality

```
jump statement:
     goto identifier ;
     continue ;
     break ;
     return expression_opt ;
```

Statements: Jump

- will cause the program flow to jump to the specified position
- **goto** is evil!
- Note: Not to be confused with the library jump functionality

```
{
bad:
     // code

     goto evil;

     // code

nasty:
     // code
     goto bad;

evil:
     // code
     goto nasty;
}
```

Statements: Jump

- will cause the program flow to jump to the specified position
- **goto** is evil!
- Note: Not to be confused with the library jump functionality

```
{
bad:
     // code

     goto evil;

     // code

nasty:
     // code
     goto bad;

evil:
     // code
     goto nasty;
}
```

```
do {
     // code
     continue;
     // code
contin:
} while (/* exp */)


do {
     // code
     goto contin;
     // code
contin:
} while (/* exp */)
```

Statements: Jump

- will cause the program flow to jump to the specified position
- **goto** is evil!
- Note: Not to be confused with the library jump functionality

```
{
bad:
     // code

     goto evil;

     // code

nasty:
     // code
     goto bad;

evil:
     // code
     goto nasty;
}
```

```
do {
     // code
     continue;
     // code
contin:
} while (/* exp */)


do {
     // code
     goto contin;
     // code
contin:
} while (/* exp */)
```

```
for (j=0; j<5; j++) {
     if (arr[j] == '\0')
          break;
     arr2[j] = arr[j];
}

switch (/* exp */) {
case 3;
     foo();
case 5:
     bar();
     break;
case 6:
     eat();
}
```

Preprocessor

- first stage in translation of program
- pulls in headers
- evaluates macros
- conditional compilation
- extras

Preprocessor

- first stage in translation of program
- pulls in headers
- evaluates macros
- conditional compilation
- extras

```
#include

#define A 5
#define B(a,b) (a+b)
#undef

#ifdef
#ifndef
#if
#else
#elif
#endif

#pragma
#error
```

Preprocessor: Header inclusion

Including system headers

- searches in a set of directories
- you can add directories to the list (with compiler switches, often `-I`)
- used for standard headers or installed libraries

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_int.h>
```

Including local headers

- starts to search from the directory of the current file

```
#include “stdio.h”
#include “foo/bar.h”
#include “../../helper/helpers.h”
```

Preprocessor: Macros

- Defining 'constants'
  - either in the code
    or
  - via the compiler (`-D`)

```
#define N 40
double arr[N];
```

- Small 'functions'
  - simplifies the code
  - facilitates the DRY principle
    (don't repeat yourself)
  - beware of side-effects!

```
#define MAX(a,b) \
    ((a > b) ? a : b)

int foo = 4;
int bar = 3;
int max = MAX(foo, bar);

int max = ((foo > bar) ? foo : bar)
```

Preprocessor: Macros

- Getting rid of macros

```
#undef N
#undef MAX
```

- Conventionally using all caps for macros

Preprocessor: Conditional Compilation

- Using macros ('defines') to only parse certain parts of a source file
  - used for optional feature of the code
  - can replace code-conditionals (theoretically faster)
  - reduces code size by only building what is needed
  - don't overdo it, it is hard to keep track of 45 different interacting options
  - **essential** to prevent multiple includes

```
#include “config.h”


#ifdef WITH_MPI          #if (defined WITH_MPI)
#  include <mpi.h>      #  include <mpi.h>
#endif                   #endif


#if (NDIM == 4)
#  define POW_NDIM(x) ((x)*(x)*(x)*(x))
#elif (NDIM == 3)
#  define POW_NDIM(x) ((x)*(x)*(x))
#elif (NDIM == 2)
#  define POW_NDIM(x) ((x)*(x))
#else
#  error NDIM
#endif
```

Preprocessor: Conditional Compilation

- Using macros ('defines') to only parse certain parts of a source file
  - used for optional feature of the code
  - can replace code-conditionals (theoretically faster)
  - reduces code size by only building what is needed
  - don't overdo it, it is hard to keep track of 45 different interacting options
  - **essential** to prevent multiple includes

```
main.c:

#include “file1.h”
#include “file2.h”


file2.h:

#include “file1.h”
```

```
in .h files:


#ifndef THIS_FILE_H
#define THIS_FILE_H

// file content

#endif
```

Preprocessor: Extras

- `#error`
  - used to stop the compilation of the code with an error
  - useful to catch incompatible compilers or incompatible defines

- `#pragma`
  - implementation specific preprocessor flags
  - used for nifty compiler specific features
  - if the used compiler does not know a given pragma statement, it will be ignored (generally producing a warning message)
  - most prominent use: OpenMP parallelizations

Preprocessor: Extras

- `#error`
  - used to stop the compilation of the code with an error
  - useful to catch incompatible compilers or incompatible defines

- `#pragma`
  - implementation specific preprocessor flags
  - used for nifty compiler specific features
  - if the used compiler does not know a given pragma statement, it will be ignored (generally producing a warning message)
  - most prominent use: OpenMP parallelizations

Preprocessor: Extras

- `#error`
  - used to stop the compilation of the code with an error
  - useful to catch incompatible compilers or incompatible defines

- `#pragma`
  - implementation specific preprocessor flags
  - used for nifty compiler specific features
  - if the used compiler does not know a given pragma statement, it will be ignored (generally producing a warning message)
  - most prominent use: OpenMP parallelizations

```
#ifdef _OPENMP
#  pragma omp parallel for
#endif
for (int i = 0; i < N; i++) {
    arr[i] = expensiveFunction(arr[i]);
}
```

Features

- The standard (C90, C99) defines a set of functions, that facilitate standard tasks (and also the headers where the functions are provided from)
  - **Input/Output:** Getting data into the code and throwing it out again
  - **Math:** implementations of standard mathematical functions
  - **Strings:** Handling of set of characters (i.e. 'strings')
  - **Memory:** Providing a framework for dynamical memory allocations
  - **and more...**
  - Full list of standard headers:
    `assert.h complex.h ctype.h errno.h fenv.h float.h inttypes.h iso646.h limits.h locale.h math.h setjmp.h signal.h stdarg.h stdbool.h stddef.h stdint.h stdio.h stdlib.h string.h tgmath.h time.h wchar.h wctype.h`
- We will only deal with a small subset of the standard functions
- A half-decent working environment will provide a complete documentation of the standard functions, e.g. in unixoid systems '`man` *`function`*'

Features

- The standard (C90, C99) defines a set of functions, that facilitate standard tasks (and also the headers where the functions are provided from)
  - **Input/Output:** Getting data into the code and throwing it out again
  - **Math:** implementations of standard mathematical functions
  - **Strings:** Handling of set of characters (i.e. 'strings')
  - **Memory:** Providing a framework for dynamical memory allocations
  - **and more...**
  - Full list of standard headers:
    ```
    assert.h complex.h ctype.h errno.h fenv.h float.h inttypes.h iso646.h limits.h
    locale.h math.h setjmp.h signal.h stdarg.h stdbool.h stddef.h stdint.h stdio.h
    stdlib.h string.h tgmath.h time.h wchar.h wctype.h
    ```
- We will only deal with a small subset of the standard functions
- A half-decent working environment will provide a complete documentation of the standard functions, e.g. in unixoid systems '`man` *`function`*'

Features

- The standard (C90, C99) defines a set of functions, that facilitate standard tasks (and also the headers where the functions are provided from)
  - **Input/Output:** Getting data into the code and throwing it out again
  - **Math:** implementations of standard mathematical functions
  - **Strings:** Handling of set of characters (i.e. 'strings')
  - **Memory:** Providing a framework for dynamical memory allocations
  - **and more...**
  - Full list of standard headers:
    ```
    assert.h complex.h ctype.h errno.h fenv.h float.h inttypes.h iso646.h limits.h
    locale.h math.h setjmp.h signal.h stdarg.h stdbool.h stddef.h stdint.h stdio.h
    stdlib.h string.h tgmath.h time.h wchar.h wctype.h
    ```
- We will only deal with a small subset of the standard functions
- A half-decent working environment will provide a complete documentation of the standard functions, e.g. in unixoid systems '`man` *`function`*'

Features

- The st ndard (C90, C99) defines a set of functions that facilitate standard tasks (
  - **Input**
  - **Math**
  - **Strin**
  - **Mem**
  - **and m**
  - Full li
    asse limits.h
    local stdio.h
    stdli

Before re-inventing the wheel,
check the standard,
ask Google (Bing, Yahoo...),
or a fellow programmer!

The functionality you look for
might be in the standard!

- We wi
- A half-decent working environment will provide a complete documentation of the standard functions, e.g. in unixoid systems '`man` *`function`*'

Input/Output

- Getting data into your code
- Writing the results to disk, report progress to user
- Concept of file descriptors (`FILE *`)
  - Three named standard ones:
    stdin: data stream from the keyboard/input redirection
    stdout: 'the screen'
    stderr: 'the screen' (but with the notion that something bad happened)
  - Files can be connected to file descriptors

Input/Output: fprintf

- Prototype

```
int fprintf(FILE *stream, const char *format, ...);
```

- Parameters
  - *stream
    The output target (stdout, stderr, or any other appropriate file handle)
  - *format
    Description of what to write out
  - …
    List of variables to write out (according to the format)

Input/Output: fprintf

- Prototype

```
int fprintf(FILE *stream, const char *format, ...);
```

- Parameters
  - *stream
    The output target (stdout, stderr, or any other appropriate file handle)
  - *format
    Description of what to write out
  - …
    List of variables to write out (according to the format)

```
fprintf(stdout, “Hello World!\n”);

int i = 42;
fprintf(stdout, “i = %i\n”, i);

double d = 1223.14451233;
fprintf(stdout, “d = %e\nd^2 = %e\n”, d, d*d);

int i = 42;
long l = (long)i;
fprintf(stdout, “i = %i\nl = %li\n”, i, l);
```

Input/Output: fprintf

- Formats

```
%i, %d   Writes an int
%e       Writes a double as [-]d.ddde±dd
%f       Writes a double as [-]ddd.ddd
%g       Selects between %e and %f depending on the number
%s       Writes a \0-terminated string
%%       Prints a %
```

- Modifiers

```
l        long int  (i.e. %li)
ll       long long int (i.e. %llu)
L        long double (i.e. %Lg)
```

Input/Output: sprintf and printf

- They work like `fprintf`, but `printf` will write to `stdout` and `sprintf` into a character array instead of a file stream.

```
int printf(const char *format, ...);

int sprintf(char *s, const char *format, ...);
```

Input/Output: (|s|f)printf return value

- The number of characters printed is returned
- In the case of errors, a negative value is returned

Input/Output: fscanf

- Prototype

```
int fscanf(FILE *stream, const char *format, ...);
```

- Parameters
  - *stream
    The input source (stdin, or any other appropriate file handle)
  - *format
    Description of what to read in
  - …
    List of pointers to variables to store the values in(according to the format)

```
int i;
fscanf(stdin, "%i", &i);

float f;
double d;
fscanf(stdin, "%f %lf", %f, &d);

int i;
long l;
fscanf(stdin, "%i %li", &i, &l);
```

Input/Output: fscanf

- Formats

```
%i, %d    Reads an int
%u        Reads an unsigned int
%f        Reads a float
```

- Modifiers

```
l         long int  (i.e. %li) or double (e.g. %lf)
```

Input/Output: sscanf and scanf

- They work like `fscanf`, but `scanf` will read from `stdin` and `sscanf` from a character array instead of a file stream.

```
int scanf(const char *format, ...);

int sscanf(char *s, const char *format, ...);
```

Input/Output: (|s|f)scanf return value

- The number of successfully matched and assigned
- That might not be equal to the number of parameters asked for

Input/Output: fopen

- Prototype

```
FILE * fopen(const char *path, const char *mode);
```

- Parameters
  - *path
    The file to open (with path, if required)
  - *mode
    The mode with which to open the file

```
FILE *f;
f = fopen("test.dat", "r");

FILE *f;
f = fopen("/data/test.dat", "w");

FILE *f;
f = fopen("../test.dat", "rb");

FILE *f;
f = fopen("run/out/test.dat", "r+");
```

Input/Output: fopen

– Modes

```
"r", "rb"     Open for reading only
              (positioned at beginning of file)
"r", "r+b"    Open for reading and writing
              (positioned at beginning of file)
"w", "wb"     Open for writing
              (file is truncated if existed)
"w+", "w+b"   Open for writing and reading
              (file is truncated if existed)
"a", "ab"     Open for appending (writing at end of file)
              (positioned at end of file)
"a+", "a+b"   Open for appending at end of file and reading
              (write position always at end of file, read position beginning of file)
```

Input/Output: fopen return value

- A file pointer providing access to the file
- If the opening failed, NULL will be returned

Input/Output: More on fopen

- There exists another function, that can change the access mode of an already available file pointer:

```
FILE * freopen(const char *path, const char *mode, FILE *f);
```

- Once the file handle is not needed anymore, the file should be closed:

```
int fclose(FILE *f);
```

Input/Output: fread/fwrite

- Prototype

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Parameters
  - *ptr: Target memory area
  - size: Number of bytes per element
  - nmeb: Number of elements to read
  - *stream: The stream from which to read

- Prototype

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- Parameters
  - *ptr: Memory area from which to copy to the file
  - size: Number of bytes per element
  - nmeb: Number of elements to read
  - *stream: The stream from which to read

Input/Output: fread/fwrite examples

```
FILE *f;
int data[48];

f = fopen(“test.dat”, “r”);
fread(data, sizeof(int), 48, f);
fclose(f);
```

```
FILE *f;
int data[48];

f = fopen(“/data/test.dat”, “w”);
fwrite(data, sizeof(int), 48, f);
fclose(f);
```

Input/Output: fread/fwrite return value

- Number of items read/written
- If errors occur, the a smaller item count (or zero)
  (error could be, e.g. end-of-file)

Math

- The functions are generally named as you would expect and do what you would guess they do

```
sin(x), cos(x), acos(x), asin(x), atan(x), tan(x)
log(x), exp(x), sqrt(x), pow(x, y)
```

- There are functions for all three types of floating point values

```
double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

Memory

- C allows for dynamic memory management
- requires `#include <stdlib.h>`
- A memory chunk can be allocated, then used and later freed
- Lifetime of allocated objects extends from allocation up to deallocation

Memory: Allocation

- Two (actually, three, see next slide) functions are available for allocation of memory:

  calloc (all bits set to zero)
  - Prototype

  - Parameters
    - nmeb: Number of elements to allocate
    - size: The size (in bytes) of one element

  malloc (unspecified initial values)
  - Prototype

  - Parameters
    - size: The number of bytes to allocate

- Both return a pointer to the lowest byte of the allocated memory region, or NULL, if no large enough contiguous memory chunk could be allocated

Memory: Allocation

- Two (actually, three, see next slide) functions are available for allocation of memory:

  calloc (all bits set to zero)
  - Prototype

```
void * calloc(size_t nmemb, size_t size);
```

  - Parameters
    - nmeb: Number of elements to allocate
    - size: The size (in bytes) of one element

  malloc (unspecified initial values)
  - Prototype

  - Parameters
    - size: The number of bytes to allocate

- Both return a pointer to the lowest byte of the allocated memory region, or NULL, if no large enough contiguous memory chunk could be allocated

Memory: Allocation

- Two (actually, three, see next slide) functions are available for allocation of memory:

  calloc (all bits set to zero)
  - Prototype

    ```
    void * calloc(size_t nmemb, size_t size);
    ```
  - Parameters
    - nmeb: Number of elements to allocate
    - size: The size (in bytes) of one element

  malloc (unspecified initial values)
  - Prototype

    ```
    void * malloc(size_t size);
    ```
  - Parameters
    - size: The number of bytes to allocate

- Both return a pointer to the lowest byte of the allocated memory region, or NULL, if no large enough contiguous memory chunk could be allocated

Memory: Allocation

- Two (actually, three, see next slide) functions are available for allocation of memory:

  calloc (all bits set to zero)
  - Prototype

    ```
    void * calloc(size_t nmemb, size_t size);
    ```
  - Parameters
    - nmeb: Number of elements to allocate
    - size: The size (in bytes) of one element

  malloc (unspecified initial values)
  - Prototype

    ```
    void * malloc(size_t size);
    ```
  - Parameters
    - size: The number of bytes to allocate

- Both return a pointer to the lowest byte of the allocated memory region, or NULL, if no large enough contiguous memory chunk could be allocated

Memory: Reallocation

- Allocated memory can be changed in size

realloc (new elements have undetermined values, old ones are kept)

- Prototype

```
void * realloc(void *ptr, size_t size);
```

- Parameters
  - *ptr: Pointer to old memory region
  - size: The new size

- if *ptr is NULL, then realloc behaves like malloc

- if *ptr is not a pointer returned by a previous call of malloc, calloc, or realloc, the behaviour is undefined

- realloc works in these steps
  - allocate new space
  - copy old data to new memory location
  - deallocate old memory

- If the new space cannot be allocated, NULL is returned and the old space is not deallocated

Memory: Reallocation

- Allocated memory can be changed in size

  realloc (new elements have undetermined values, old ones are kept)
  - Prototype

```
void * realloc(void *ptr, size_t size);
```

  - Parameters
    - *ptr: Pointer to old memory region
    - size: The new size
  - if *ptr is NULL, then realloc behaves like malloc
  - if *ptr is not a pointer returned by a previous call of malloc, calloc, or realloc, the behaviour is undefined
  - realloc works in these steps
    - allocate new space
    - copy old data to new memory location
    - deallocate old memory
  - If the new space cannot be allocated, NULL is returned and the old space is not deallocated

Memory: Reallocation

- Allocated memory can be changed in size

realloc (new elements have undetermined values, old ones are kept)

- Prototype

```
void * realloc(void *ptr, size_t size);
```

- Parameters
  - *ptr: Pointer to old memory region
  - size: The new size

- if *ptr is NULL, then realloc behaves like malloc

- if *ptr is not a pointer returned by a previous call of malloc, calloc, or realloc, the behaviour is undefined

- realloc works in these steps
  - allocate new space
  - copy old data to new memory location
  - deallocate old memory

- If the new space cannot be allocated, NULL is returned and the old space is not deallocated

Memory: Reallocation

- Allocated memory can be changed in size

  realloc (new elements have undetermined values, old ones are kept)
  - Prototype

```
void * realloc(void *ptr, size_t size);
```

  - Parameters
    - *ptr: Pointer to old memory region
    - size: The new size
  - if *ptr is NULL, then realloc behaves like malloc
  - if *ptr is not a pointer returned by a previous call of malloc, calloc, or realloc, the behaviour is undefined
  - realloc works in these steps
    - allocate new space
    - copy old data to new memory location
    - deallocate old memory
  - If the new space cannot be allocated, NULL is returned and the old space is not deallocated

Memory: Reallocation

- Allocated memory can be changed in size

  realloc (new elements have undetermined values, old ones are kept)
  - Prototype

```
void * realloc(void *ptr, size_t size);
```

  - Parameters
    - *ptr: Pointer to old memory region
    - size: The new size
  - if *ptr is NULL, then realloc behaves like malloc
  - if *ptr is not a pointer returned by a previous call of malloc, calloc, or realloc, the behaviour is undefined
  - realloc works in these steps
    - allocate new space
    - copy old data to new memory location
    - deallocate old memory
  - If the new space cannot be allocated, NULL is returned and the old space is not deallocated

Memory: Deallocation

- Return memory chunk back to the system for other usage

free

- Prototype

```
void free(void *ptr);
```

- Parameters
  - *ptr: Pointer to memory region that should be freed

- *ptr **must** be a pointer returned by a previous call of malloc, calloc, or realloc

- *ptr may be NULL, in which case no operation is performed

Memory: Deallocation

- Return memory chunk back to the system for other usage

free

- Prototype

```
void free(void *ptr);
```

- Parameters
  - *ptr: Pointer to memory region that should be freed

- *ptr **must** be a pointer returned by a previous call of malloc, calloc, or realloc

- *ptr may be NULL, in which case no operation is performed

Memory: Deallocation

- Return memory chunk back to the system for other usage

free

- Prototype

```
void free(void *ptr);
```

- Parameters
  - *ptr: Pointer to memory region that should be freed

- *ptr **must** be a pointer returned by a previous call of malloc, calloc, or realloc

- *ptr may be NULL, in which case no operation is performed

Memory: Pitfalls

- not dealing with NULLs
- the size is precious
- double free corruptions
- memory leaks

```c
double *data;
uint64_t num = 1L << 50; // 1024TB

data = malloc(sizeof(double) * num); // returns NULL

for (uint64_t i = 0; i < num; i++)
    data[i] = (double)i; // Produces a segmentation fault
```

Memory: Pitfalls

- not dealing with NULLs
- the size is precious
- double free corruptions
- memory leaks

```
int
f(double *data, int numElements)
{
     for (int i = 0; i < numElements; i++)
          data[i] = exp(data[i]);
}
```

Memory: Pitfalls

- not dealing with NULLs
- the size is precious
- double free corruptions
- memory leaks

```
int
f(double *data, int numElements)
{
    for (int i = 0; i < numElements; i++)
        printf("%15.10f\n", data[i]);
    free(data);
}

// ...

    f(data, numElements);
    free(data);  // Black dragons...

// ...
```

Memory: Pitfalls

- not dealing with NULLs
- the size is precious
- double free corruptions
- memory leaks

```
void
leakingMemory(void)
{
     double *data = malloc(sizeof(double) * 1024);

     return;
}
```

Error handling

Basically falls under 'best practise' but there are two noteworthy things provided by the library:

- <errno.h> provides a error variable that may be set by several functions

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

if (fclose(stdout) != 0) {
    int errnum = errno;
    fprintf(stderr, "%s", strerror(errnum));
    exit(EXIT_FAILURE);
}
```

- <assert.h> provides a macro to do hard runtime checks

Error handling

Basically falls under 'best practise' but there are two noteworthy things provided by the library:

- <errno.h> provides a error variable that may be set by several functions

```c
#include <errno.h>
#include <string.h>
#include <stdlib.h>

if (fclose(stdout) != 0) {
    int errnum = errno;
    fprintf(stderr, "%s", strerror(errnum));
    exit(EXIT_FAILURE);
}
```

- <assert.h> provides a macro to do hard runtime checks

```c
#include <assert.h>

int
f(double *a, int n)
{
    assert(a != NULL);
    assert(n > 0 && n < 1024);
    assert(1 == 0); // Aborts code and produces a core file
```

Files

Two types of (plain text) files:

- Header Files (*.h)
  - **declare** things that can be used
- Source Files (*.c)
  - **implement** things

Generated (binary) files:

- Object files (*.o)
  - contains the compiled code
- Libraries (lib*.a, lib*.so, *.dll, …)
  - collection of object files
- Executable (no specific ending)
  - Can be executed

Files

Two types of (plain text) files:

- Header Files (*.h)
  - **declare** things that can be used
- Source Files (*.c)
  - **implement** things

Generated (binary) files:

- Object files (*.o)
  - contains the compiled code
- Libraries (lib*.a, lib*.so, *.dll, …)
  - collection of object files
- Executable (no specific ending)
  - Can be executed

main.c xmem.h xmem.c

```c
/*--- Includes ----------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <math.h>
#include “xmem.h”


/*--- M A I N -----------------------------------------------------------*/
int
main(int argc, char **argv)
{
    uint64_t numDataPoints = 4294967295; // 2^32 - 1
    double   *data         = xmalloc(sizeof(double)*(numDataPoints));

    for (uint64_t i = 0; i < numDataPoints; i++)
        data[i] = sqrt((double)i);

    xfree(data);

    return EXIT_SUCCESS;
}
```

main.c xmem.h xmem.c

```c
#ifndef XMEM_H
#define XMEM_H


/*--- Includes ------------------------------------------------------------*/
#include <stdlib.h>


/*--- Exported global variables -------------------------------------------*/
extern size_t global_bytesAllocated;


/*--- Prototypes of exported functions ------------------------------------*/
extern void *
xmalloc(size_t size);

extern void
xfree(void *ptr);


#endif
```

main.c xmem.h xmem.c

```c
/*--- Includes ------------------------------------------------------------*/
#include "xmem.h"
#include <stdio.h>


/*--- Implementations of exported variables -----------------------------*/
size_t global_bytesAllocated = 0;


/*--- Implementations of exported functions -----------------------------*/
extern void *
xmalloc(size_t size)
{
    void *ptr;

    ptr = malloc(size);
    if (ptr == NULL) {
        fprintf(stderr, "Failed to allocate %zi bytes\n", size);
        exit(EXIT_FAILURE);
    }
    global_bytesAllocated += size;

    return ptr;
}

extern void
xfree(void *ptr)
{
    if (ptr != NULL) {
        free(ptr);
        global_bytesAllocated -= size;
    }
}
```

Compiling

'Compiling the code' generally means:

- translating all .c files to .o files
- linking the .o files (and external libraries) together, thereby producing an executable

Compiling: Translating

For gcc:

```
export CC=gcc

$(CC) -std=c99 -Wall -c -o main.o main.c
$(CC) -std=c99 -Wall -c -o xmem.o xmem.c
```

Compiler used here: Gnu C Compiler (gcc)

Flags:

- -c: Tells to compiler to produce an object file
- -o: Specifies the filename of the output file
- -std: Select the C standard (here: C99)
- -W: Specifying compiler warning (here: all)

Compiling: Translating

For gcc:

```
export CC=gcc

$(CC) -std=c99 -Wall -c -o main.o main.c
$(CC) -std=c99 -Wall -c -o xmem.o xmem.c
```

Compiler used here: Gnu C Compiler (gcc)

Flags:

- -c: Tells to compiler to produce an object file
- -o: Specifies the filename of the output file
- -std: Select the C standard (here: C99)
- -W: Specifying compiler warning (here: all)

Compiling: Linking

For gcc:

```
export CC=gcc

$(CC) -o myProgram main.o xmem.o -lm
```

Compiler used here: Gnu C Compiler (gcc)

Flags:

- -o: Specifies the filename of the output file
- -l: Linking a library (here: -lm, linking the math library)

Compiling: Linking

For gcc:

```
export CC=gcc

$(CC) -o myProgram main.o xmem.o -lm
```

Compiler used here: Gnu C Compiler (gcc)

Flags:

- -o: Specifies the filename of the output file
- -l: Linking a library (here: -lm, linking the math library)

Compiling: Translating & Linking

For simple one file programs it is more convenient to directly produce the binary without first generating the object file:

```
export CC=gcc

$(CC) -std=c99 -Wall -o myProgram mySourceCode.c -lm
```

Makefiles

- Building project with more than one file tends to be tedious if done by hand
- `make` is a utility that can automate the compilation
- this requires a Makefile that describes the dependencies of the source files (that file should be names `Makefile` or `makefile`)

Makefiles: Simple Example

Instead of

```
export CC=gcc

$(CC) -std=c99 -Wall -c -o main.o main.c
$(CC) -std=c99 -Wall -c -o xmem.o xmem.c
$(CC) -o myProgram main.o xmem.o -lm
```

simply

```
make myProgram
```

with

Makefile

```
CC = gcc
CFLAGS = -std=c99 -Wall

myProgram: main.o xmem.o
    $(CC) -o myProgram main.o xmem.o -lm
```

Makefiles: Structure

Generally, makefiles consist of a list of rules of the form

```
target: prerequisites
    command1
    command2
```

- Note that the commands must be intended with a tab!
- make knows a few rules a-priori, especially, it knows how to generate object files from source files (hence there was no need to specify a rule how to generate main.o and xmem.o in the previous example).
- It is possible to generate complex dependencies (e.g. a.c needs to be recompiled, because b.h changed) on the fly with pattern rules (see `info make` for more details).

Makefiles: Complex Example

```
CC=gcc
DEPCC=gcc
CFLAGS=-std=c99 -Wall -O3 -fopenmp
CPPFLASG=-I/opt/fftw/include/
LDFLAGS=-L/opt/fftw/lib/
LIBS=-lfftw -lm

.PHONY: all clean

progName = myProg

sources = main.c $(progName).c read.c write.c work.c

%.d: %.c
    @set -e; rm -f $@; \
         $(DEPCC) -MM $(CPPFLAGS) $< > $@.$$$$; \
         sed 's,\($*\)\.o[ :]*,\1.o $@ : ,g' < $@.$$$$ > $@; \
         rm -f $@.$$$$

all:
    $(MAKE) $(progName)

clean:
    rm -f $(progName) $(sources:.c=.o)

$(progName): $(source:.c=.o)
    $(CC) $(LDFLAGS) $(CFLAGS) -o $(progName) $(sources:.c=.o) $LIBS


-include $(sources:.c=.d)
```

Debugging

First rule of debugging:

Read compiler error messages.

Second rule of debugging:

Read compiler warning messages.

Methods of debugging:

- **printf-statements**
  used to figure out at what point the code breaks and to print out values of possibly affected variables
- **gdb**
  interactive way to follow to program flow with complete access to all variables and the complete stack
- **valgrind**
  used to catch errors in memory handling (memory leaks, wrong access, undefined values)

Debugging

First rule of debugging:

Read compiler error messages.

Second rule of debugging:

Read compiler warning messages.

Methods of debugging:

- **printf-statements**
  used to figure out at what point the code breaks and to print out values of possibly affected variables

- **gdb**
  interactive way to follow to program flow with complete access to all variables and the complete stack

- **valgrind**
  used to catch errors in memory handling (memory leaks, wrong access, undefined values)

Debugging

First rule of debugging:

Read compiler error messages.

Second rule of debugging:

Read compiler warning messages.

Methods of debugging:

- **printf-statements**
  used to figure out at what point the code breaks and to print out values of possibly affected variables

- **gdb**
  interactive way to follow to program flow with complete access to all variables and the complete stack

- **valgrind**
  used to catch errors in memory handling (memory leaks, wrong access, undefined values)

Debugging

First rule of debugging:

Read compiler error messages.

Second rule of debugging:

Read compiler warning messages.

Methods of debugging:

- **printf-statements**
  used to figure out at what point the code breaks and to print out values of possibly affected variables
- **gdb**
  interactive way to follow to program flow with complete access to all variables and the complete stack
- **valgrind**
  used to catch errors in memory handling (memory leaks, wrong access, undefined values)