

# Compiling and Linking C / C++ Programs

Instructor: Peter Baumann

email: [p.baumann@jacobs-university.de](mailto:p.baumann@jacobs-university.de)

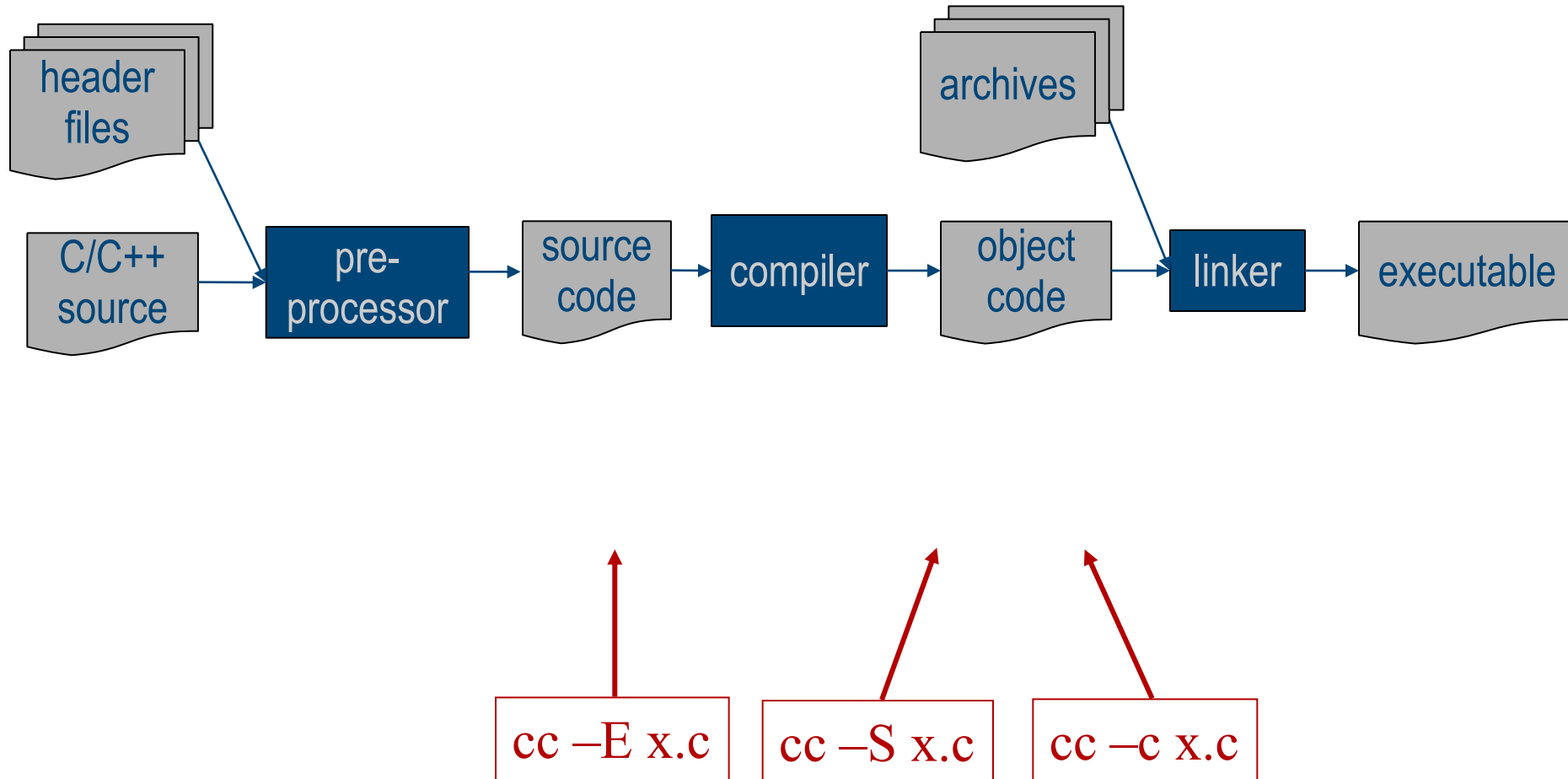
tel: -3178

office: room 88, Research 1



- ...watch your code like you never have seen it before!

# Compile/Link Steps Overview



# File Extension Conventions

- C source code .c
- C include file .h
- C++ source file .cc , .C, .cxx, .c++, .cp, .cpp
- C++ header file .hh, .hpp
- Object file (relocatable) .o
- Executable no extension (Windows: .com, .exe)
- Library
  - static .a
  - dynamic .so

# The C Preprocessor

- Purpose:
  - Define commonly used constants, code fragments, etc.
  - Conditional compilation (code activation depending on external settings)

- Main mechanism: replace by **textual substitution**
  - No idea about semantics (parentheses, semicolons, ...) !!
  - Does not follow C syntax

```
#define X 1
```

```
const int x = 1;
```

- Preprocessor directives
  - `#include`
  - `#define`
  - `#if` / `#ifdef`
  - ...plus more

# Using Preprocessor Directives

- Conditional compilation

- Include guard in header files, eg in `mystdio.h`:

```
#ifndef _MYSTDIO_H_
#define EOF (-1)
#define NULL 0L
#define _MYSTDIO_H_
#endif _MYSTDIO_H_
```

- Include files

- `#include <stdio.h>` – taken from predefined location
- `#include "myclass.h"` – taken from local directories

- Where to find include files?

- Standard locations: `/usr/include, /usr/local/include, ...`
- Specified locations `cc -I/home/project/include`

- Can also pass definitions

- `cc -DCOMPILE_DATE=\"`date`\" -DDEBUG`

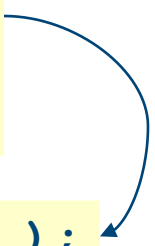
# Common Preprocessor Pitfall

- Use parentheses!!!

- bad:

```
#define mult(a,b) a*b
main()
{
printf( "(2+3)*4=%d\n", mult(2+3,4) );
}
```

printf( "(2+3)\*4=%d\n", 2+3\*4 );



- good:

```
#define mult(a,b) ((a)*(b))
main()
{
printf( "(2+3)*4=%d\n", mult(2+3,4) );
}
```

printf( "(2+3)\*4=%d\n", ((2+3)\*(4)) );



# The C(++) Compiler

- Task: Generate (relocatable) **machine** („object“) **code** from source code
- **Relocation**: code can sit at different places in address space
- Address space classified into „segments“
  - Code, text, data, ...
- Note: OS (with HW support) uses this to implement **user address space**
  - Actual main memory address = base address + relative address
  - Base address kept in segment register, added dynamically by CPU
  - Security: program cannot access base register ("privileged mode"), hence cannot address beyond its segment limits



# Object Files

- Contain code for a program fragment (module)
  - Machine code, constants, size of static data segments, ...

- `$ nm rserver_main.o`

```
00000000c D clientTimeout
          U __cxa_allocate_exception
          U __cxa_begin_catch
          U __cxa_end_catch
          U __cxa_free_exception
          U __cxa_throw
000000004 B debugOutput
          U free
          U getenv
00000120 B globalHTTPPort
00000000 T main
          U memset
```

or `objdump`

# External Functions & Variables

- Module **server**:  
Variable **sema** allocated in data segment
- Module **client**:  
functions obtain sema address by
  - Module **server** offset  
+ local address **sema**
- Cross-module addressing rules:
  - (no modifier) = locally allocated, globally accessible
  - static = locally allocated, locally accessible
  - extern = allocated in other compilation unit
- Why is this wrong?
  - **extern int sema = 1;**

```
int sema = 0;

int serverBlock()
{
    if (sema==0)
        sema = 1;
    return sema;
}
```

```
extern int sema;

int clientBlock()
{
    if (sema==0)
        sema = 1;
    return sema;
}
```

- Problem: classes convey complex naming, not foreseen in classic linkage
  - Classes, overloading, name spaces, ...
    - Ex: `MyClass1::myFunc()`  
`MyClass2::myFunc()`
  - But only named objects in files, flat namespace
- Solution: **name mangling**
  - Compiler **modifies names** to make them unique (prefix/suffix)
  - Ex: `Transaction::begin()`  
→ `_ZN13r_Transaction5beginENS_8r_TAModeE`
- Every compiler has its individual mangling algorithm!
  - Code compiled with different compilers is **incompatible**

# Name Mangling (contd)

- Q: My linker cannot find function `flatFunc()`, although I link against library `libff.a` which definitely contains `flatFunc()`.
  - Proof: `nm libff.a | grep flatFunc`
- A: You're probably linking with C code.  
Tell the C++ compiler that it's C, not C++, to avoid name mangling:

```
extern "C"  
{  
    void flatFunc();  
}
```

# The Linker/Loader

- Task: generate *one* executable file from *several* object and library files
  - Read object files
  - Resolve addresses from (relocatable) code
  - Link runtime code (start/exit handling!)
  - Add all code needed from libraries
  - If required: establish stubs to dynamic libraries
  - Write executable code into file, set magic number, etc.
- cc, g++, etc. have complex logics inside
  - can silently *invoke* linker, don't link themselves!
  - Common shorthand: `cc -o x x.c`
- Ex: `ld -o x /lib/crt0.o x.o -lc`

*John R. Levine:  
Linkers and Loaders.  
Morgan Kaufmann, 1999*

# What It Really Looks Like

```
if (DEBUG_LEVEL >= DEBUG_MEM) {
    memrec_add_var(&malloc_rec, filename, line, temp, size);
}
return (temp); void *ptr, size_t
size_t count;
realloc(const char *var, const char *filename, unsigned long line, void *ptr, size_t size)
__LINE__
void *temp;
unsigned long
MALLOCCALL_DEBUG
++realloc_count;
if (!(realloc_count % REALLOC_MOD)) {
    D_MEM(("Calls to realloc(): %d\n", realloc_count));
}
endif
line, size_t count;
D_MEM(("Variable %s (%8p -> %lu) at %s:%lu\n", var, ptr, (unsigned long) size, filename, line));
if (ptr == NULL) {
    MODtemp = (void *) libast_malloc(__FILE__, __LINE__, size);
} else {
    temp = (void *) realloc(ptr, size);
    ASSERT_RVAL(temp != NULL, ptr);
    if (DEBUG_LEVEL >= DEBUG_MEM) {
        memrec_chg_var(&malloc_rec, var, filename, line, ptr, temp, size);
        memrec_add_var(&malloc_rec, filename, line, temp, size);
    }
}
return (temp);
const char *filename, unsigned long line, size_t count, size_t size)
```

**CC -V x.c**

```
cc1 ... x.c ... -o /tmp/ccWs4dqa.s
as ... -o /tmp/cckBDoD2.o /tmp/ccWs4dqa.s
collect2 ... -o x /lib/ld-linux.so.2 \\
crt1.o crti.o crtbegin.o /tmp/cckBDoD2.o
-lgcc -lgcc_eh -lc -lgcc -lgcc_eh
crtend.o crtn.o
```

- By default, executable contains symbol tables
  - Function names, addresses, parametrization
  - Static variables
  - ...some more stuff
- Disadvantages:
  - Allows reverse engineering (gdb!)
  - Substantially larger code files
- Before shipping: strip executables
  - **file rasserver**  
`rasserver: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), not stripped`
  - **strip rasserver**

# Libraries (Archive Files)

- Library = archive file containing a collection of object files
  - Code fragments (classes, modules, ...)
  - `ar rv libxxx.a file1.o file2.o ...`
- Object files vs. Libraries
  - Object file linked in **completely**, from library **only what is actually needed**
- Static vs. Dynamic
  - **Static library**: code is added to the executable, just like object file; not needed after linkage
  - **Dynamic library**: only stub linked in, runtime system loads; needed at runtime (version!)
- Naming conventions (Linux)
  - Static libraries: **libxxx.a**
  - Dynamic libraries: **libxxx.so**
  - link with: `ld ... -lxxx`



- How to **find** my dynamic libraries?
  - `LD_LIBRARY_PATH` variable, similar to `PATH`: set before program start
- How to **know about use** of dynamic libraries?
  - `$ ldd rasserver`

```
linux-gate.so.1 => (0xfffffe000)
libstdc++.so.5 => /usr/lib/libstdc++.so.5 (0x40028000)
libm.so.6 => /lib/tls/libm.so.6 (0x400e5000)
libgcc_s.so.1 => /lib/libgcc_s.so.1 (0x40128000)
libc.so.6 => /lib/tls/libc.so.6 (0x40130000)
libresolv.so.2 => /lib/libresolv.so.2 (0x4029c000)
```

# Schematic Program Run

## ■ OS:

- Open file
- Look at first page:  
magic number, segment sizes, etc.
- Allocate segments  
(code, runtime stack, heap, ...)
- Read code file into code segment
- Set up process descriptor  
(external resources, limits, ...)
- Pass control to this process
- Handle system calls
- Terminate program,  
free process slot and resources

## ■ Application program:

- Set up runtime environment  
(**argv/argc**, ...)
- Call **main()**
- On system calls, interrupts, etc.:  
pass control to OS
- Upon **exit()**,  
or **main()**'s **return**,  
or a forced abort:  
clean up (close file descriptors, sockets, ...),  
pass back to OS

# Summary

- To create executable program, you must perform:
  - Preprocess – textually expands definitions, condition-guarded code pieces
  - Compile – translates source code into relocatable machine code („object code“)
  - Link – bind object files and archives into executable program

```
cc -o x x.c
```

=

```
cpp x.c x.cpp  
cc -o x.o -c x.cpp  
ld -o x /lib/crt0.o x.o -lc
```

# Summary (contd.)

- A word about code quality
  - Set compiler to max rigidity: `cc -W -Wall ...`
  - Eliminate *all* warnings
- Finally, the formatting war:

## ANSI C:

```
void foo()  
{  
    myAction();  
}
```

## Kernighan/Ritchie:

```
void foo() {  
    myAction();  
}
```

- The answer: whatever style, *use one coherently*
  - Use automatic beautifier (see my SE course page for some)