Advanced C++

4. Compiling, linking and execution

14.11.2017

Sidnev A.A.

Compilers, Assemblers and Linkers

1. **Preprocessor**
   - include-files
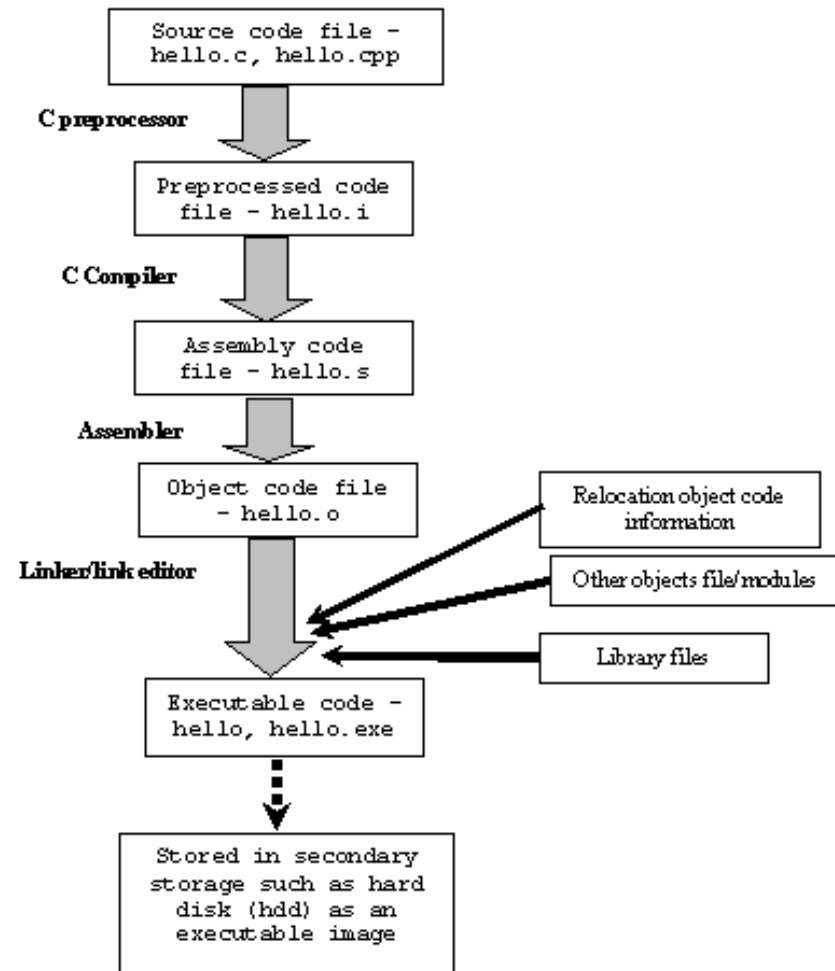   - conditional compilation instructions
   - macros

2. **Compiler**
   - const/constexpr

3. **Assembler**

4. **Linker**


```
Source code file -
hello.c, hello.cpp
```
C preprocessor
```
Preprocessed code
file - hello.i
```
C Compiler
```
Assembly code
file - hello.s
```
Assembler
```
Object code file
- hello.o
```
```
Relocation object code
information
```
Linker/link editor
```
Other objects file/modules
```
```
Library files
```
```
Executable code -
hello, hello.exe
```
```
Stored in secondary
storage such as hard
disk (hdd) as an
executable image
```

http://www.tenouk.com/ModuleW.html

Object file

Section	Short description
.text	Executable instruction codes, shared among every process running the same binary. READ/EXECUTE.
.bss	Block Started by Symbol. It holds un-initialized global and static variables. Doesn't take up any actual space in the object file.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. READ/WRITE.
.rdata	Also known as .rodata (read-only data) section. This contains constants and string literals.
.reloc	Stores the information required for relocating the image while loading.
Symbol table	Symbol table holds information needed to locate and relocate a program's symbolic definitions and references.
Relocation records	Relocation records are information used by the linker to adjust section contents.

Linker

- Most assembly instructions are easily translated into machine code using a one-to-one correspondence
- But in our program we declared **labels** for addresses
  - Addresses in the .bss and the .data segments
  - Addresses in the .text segments (for jumps)
- **Question**: How should the assembler translate instructions that use these labels into machine code?
  - E.g., add [L], ax
  - E.g., call my_function
- **Answer**: it cannot do the full job without knowing the "whole" program so as to determine addresses

Relocation records

objdump -r test.o

Symbol tables

objdump -t test.o

http://www.tenouk.com/ModuleW.html
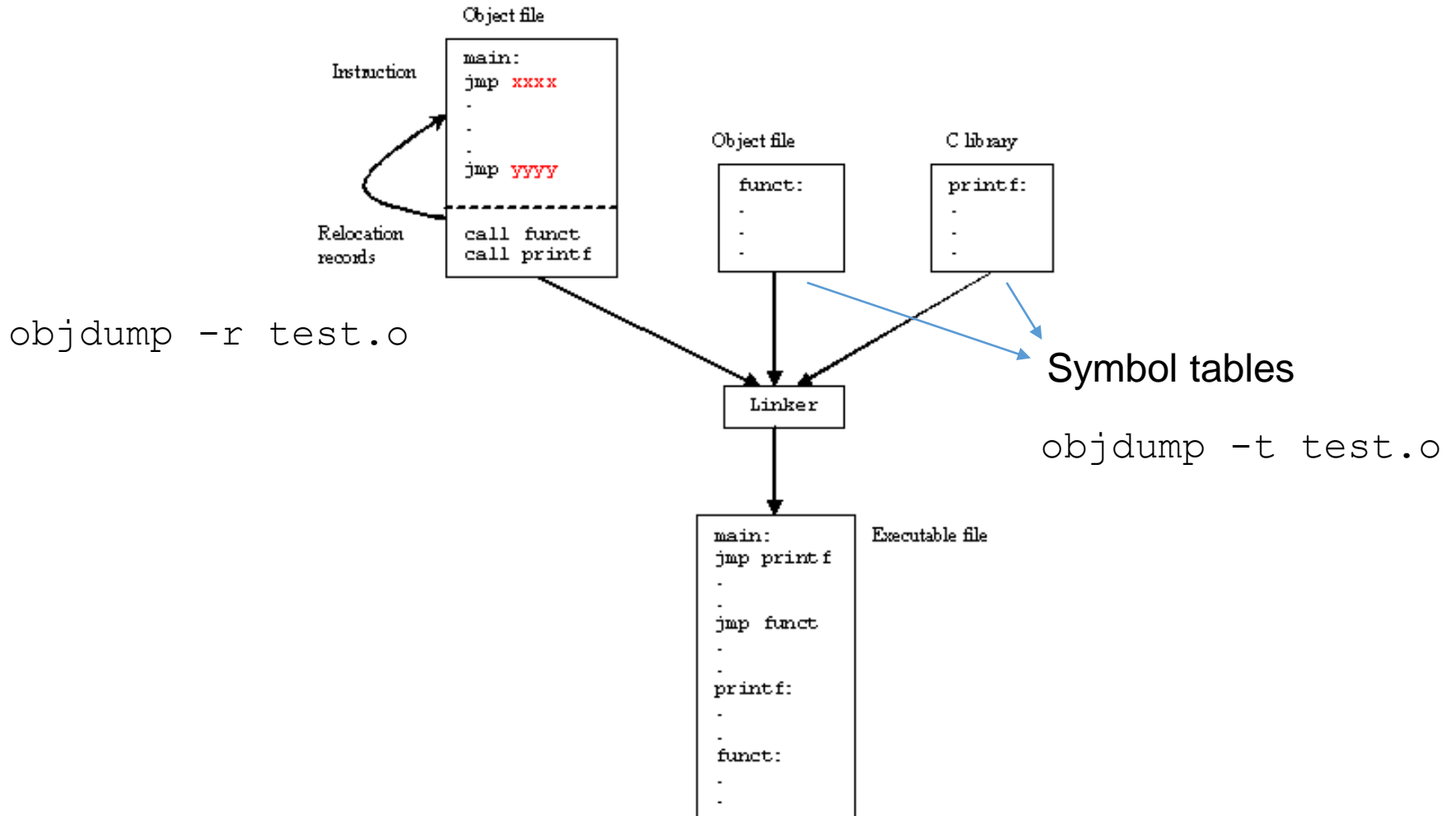
objdump (old example)

```cpp
// A.cpp -> A.obj
#include <iostream>
using std::cout;
using std::endl;

namespace A {
int x1;
int x2 = 1;
extern int x3;
extern int x4;
extern const int x5;

int f() {
  cout << x1 << endl;
  cout << x2 << endl;
  cout << x3 << endl;
  cout << x4 << endl;
  cout << x5 << endl;
  return 5;
}
}  // namespace A
```

```cpp
// B.cpp -> B.obj
float x1;

namespace A {
int x3 = 2;
int x4;
extern const int x5 = 1;

int f();
}  // namespace A

int main() {
  x1 = A::f();
}
```

objdump (symbol table)

// A.o

```
0000000000000000 g     O .bss
0000000000000004 _ZN1A2x1E

0000000000000000 g     O .data
0000000000000004 _ZN1A2x2E

0000000000000000 g     F .text
00000000000000a6 _ZN1A1fEv
```

// B.o

```
0000000000000000 g     O .bss
0000000000000004 x1

0000000000000000 g     O .data
0000000000000004 _ZN1A2x3E

0000000000000004 g     O .bss
0000000000000004 _ZN1A2x4E

0000000000000000 g     O .rodata
0000000000000004 _ZN1A2x5E

0000000000000000 g     F .text
0000000000000020 main

0000000000000000        *UND*
0000000000000000 _ZN1A1fEv
```

objdump (relocation records)

// A.o

OFFSET TYPE   VALUE

0000000000000006 R_X86_64_PC32
_ZN1A2**x1**E-0x0000000000000004

000000000000000d R_X86_64_32
_ZSt4**cout**

0000000000000025 R_X86_64_PC32
_ZN1A2**x2**E-0x0000000000000004

000000000000002c R_X86_64_32
_ZSt4**cout**

…

// B.o

OFFSET TYPE   VALUE

0000000000000005 R_X86_64_PC32
_ZN1A1**f**Ev-0x0000000000000004

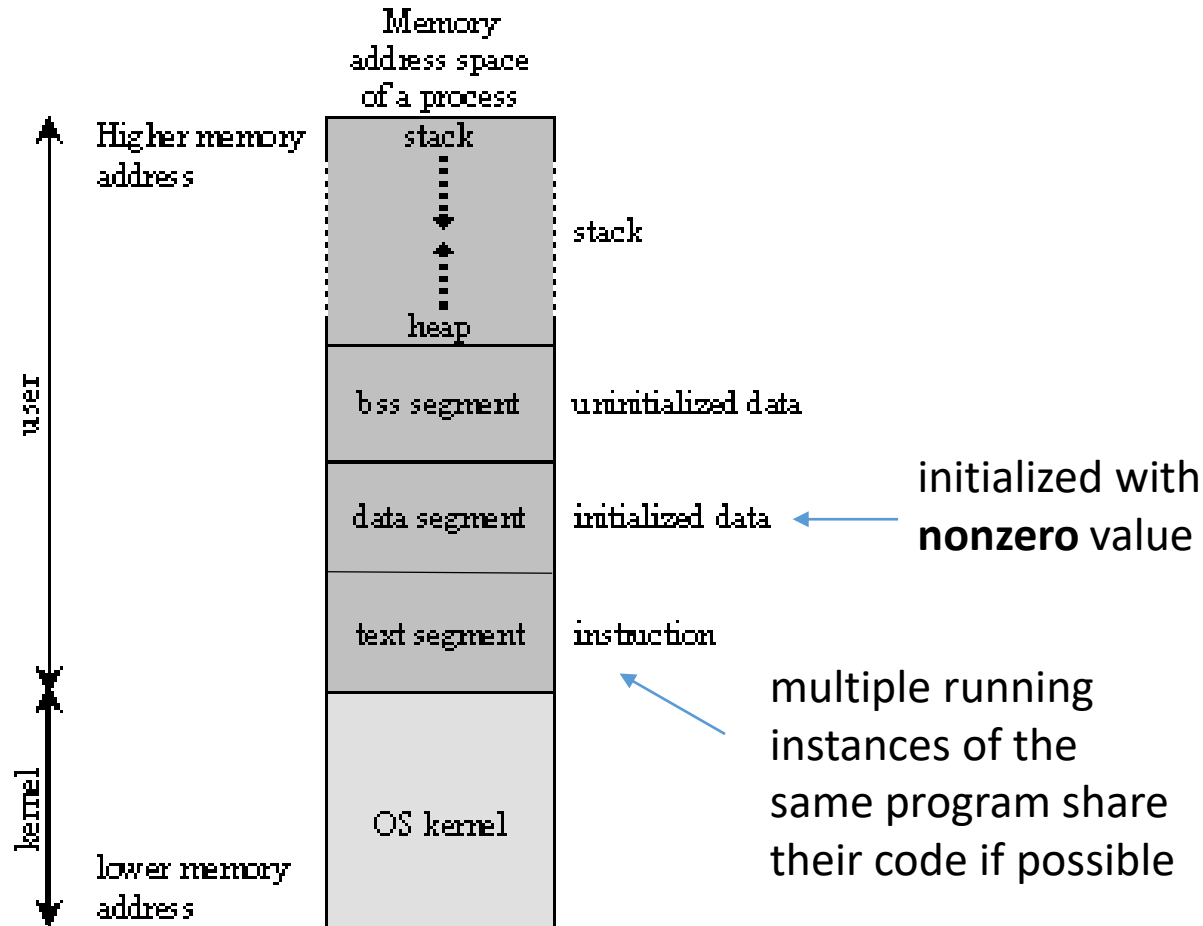0000000000000015 R_X86_64_PC32 **x1**-
0x0000000000000004

Linker

- Step 1: concatenate all the text segments from all the .o files

- Step 2: concatenate all the data/bss segments from all the .o files

- Step 3: Resolve references
  - "symbol not found"
  - "multiply defined"

The loader

- Read the executable file's header to find out the size of the text and data segments

- Creates a new address space for the program that is large enough to hold the text and data segments, and to hold the stack (within some bounds)

- Copies the text and data segments into the address space

- Copies arguments passed to the program on the stack. Initializes the registers
  - Clear most of them, set ESP to the top of the stack

- Jump to a standard "start up routine", which sets the PC and calls the exit() system call when the program terminates

Loading executable

Memory
address space
of a process

Higher memory
address

stack

stack

heap

user

bss segment — uninitialized data

data segment — initialized data ← initialized with **nonzero** value

text segment — instruction

multiple running
instances of the
same program share
their code if possible

kernel

OS kernel

lower memory
address

http://www.tenouk.com/ModuleW.html

Execution

Higher memory address

Stack	System
	env argv argc
	Auto variables for main()	← main() frame pointer (EBP)
	Auto variable for func()
	Available for stack growth	← Stack pointer (ESP), points at the top of the stack - grows downward
Shared libraries	malloc.o (lib*.so)
	printf.o (lib*.so)	Library functions if dynamically linked – the usual case
	Available for heap growth
		← brk() point
	Heap (malloc(), calloc(), new)

data	Global variables	Uninitialized data - bss
	int y = 100;	Initialized data - data
Text (Compiled code, a.out)	malloc.o (lib*.a)
	printf.o (lib*.a)	Library functions if statically linked – not the usual case
	file.o
	main.o func()	← The return address
	crt0.o (startup routine)

Lower memory address

http://www.tenouk.com/ModuleW.html