Incremental Linking with Gold

Linux Foundation Collaboration Summit
April 5, 2012

Cary Coutant

About the Gold Linker

ELF-only, open-source linker, by Ian Lance Taylor

- Written in C++
- Supports i386, x86-64, ARM, and SPARC targets (PowerPC under development)
- Significantly faster than the old GNU linker

The Edit-Compile-Debug Cycle

```
$ make
g++ -c -g a.cc
g++ -c -g b.cc
…
g++ -c -g z.cc
g++ a.o b.o … z.o
```

Initial compile: hours

Initial link: 30 sec.

```
$ … test …
$ vi b.cc
$ make
g++ -c -g b.cc
g++ a.o b.o … z.o
```

Recompile: 2 sec.

Relink: 30 sec.

```
$ … test …
```

With Incremental Linking

```
$ make LDFLAGS=-Wl,--incremental
g++ -c -g a.cc
g++ -c -g b.cc
…
g++ -c -g z.cc
g++ LDFLAGS=-Wl,--incremental …
```

Initial compile: hours

Initial link: 20 sec.

```
$ … test …

$ vi b.cc

$ make LDFLAGS=-Wl,--incremental
g++ -c -g b.cc
g++ LDFLAGS=-Wl,--incremental …
```

Recompile: 2 sec.

Relink: 3 sec.

```
$ … test …
```

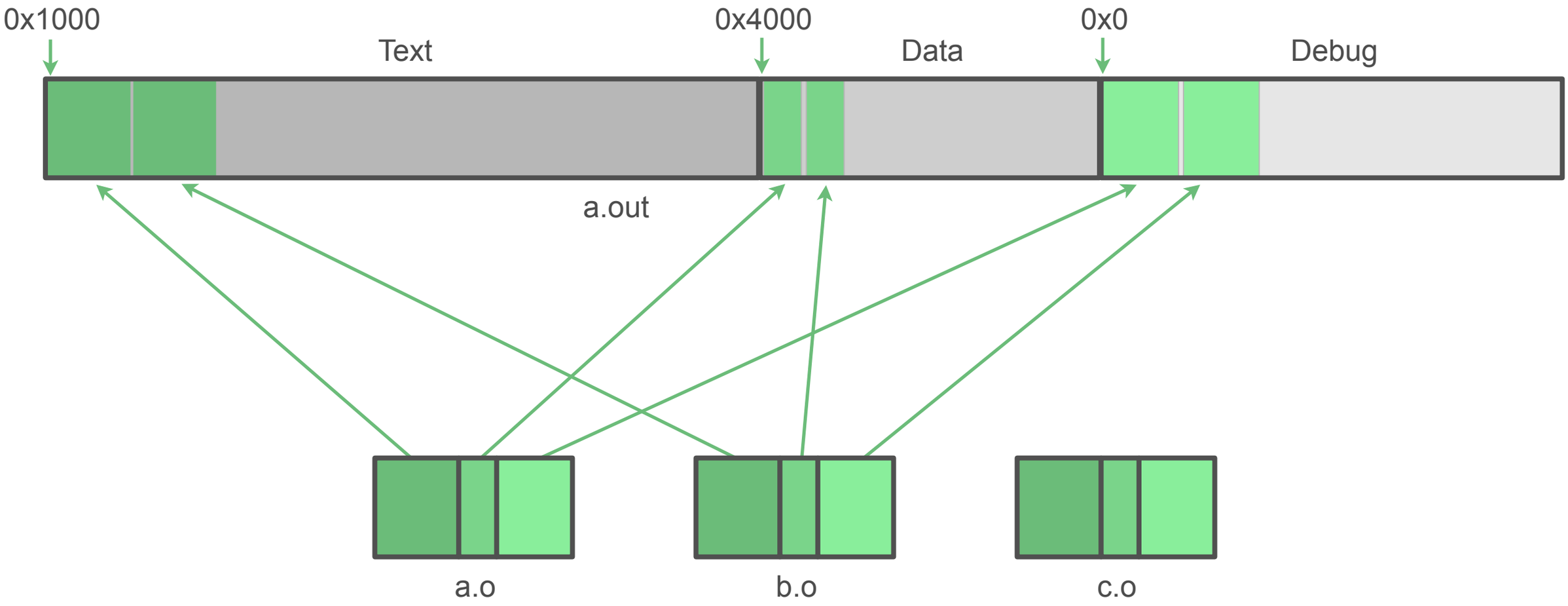
Fundamentals of a Linker

1. Combine object files, grouping by section.

2. Resolve symbolic references between files.

3. Relocate code and data.

Combine



Resolve

a.c

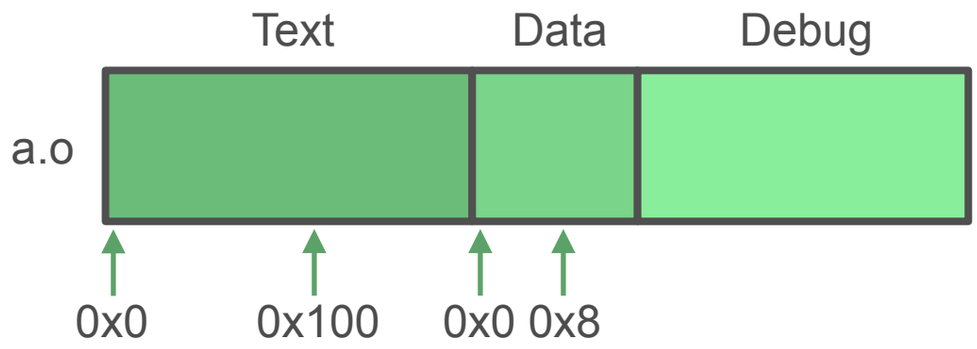
```
int var1 = 1;
int *var2 = &var1;

int function1()
{
  int i;
  i = function2();
  i += var1;
  return function3(i);
}

int function2()
{
  return *var2;
}
```

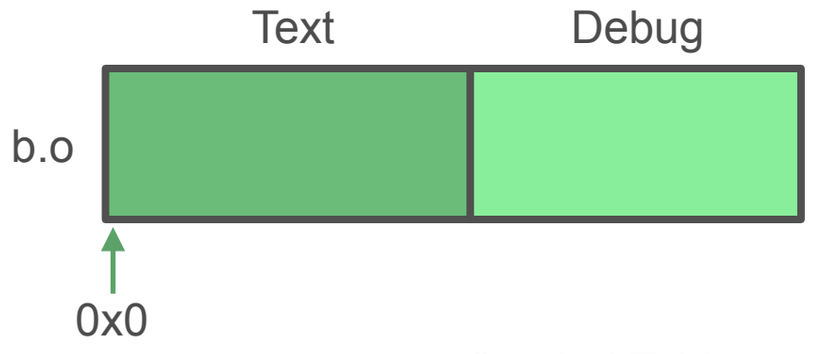
b.c

```
int function3(int x)
{
  return x * 5;
}
```



Symbol Table

Symbol	Section	Offset
function1	Text	0x0
function2	Text	0x100
var1	Data	0x0
var2	Data	0x8
function3	Undefined	



Symbol Table

Symbol	Section	Offset
function3	Text	0x0

Relocate

a.c

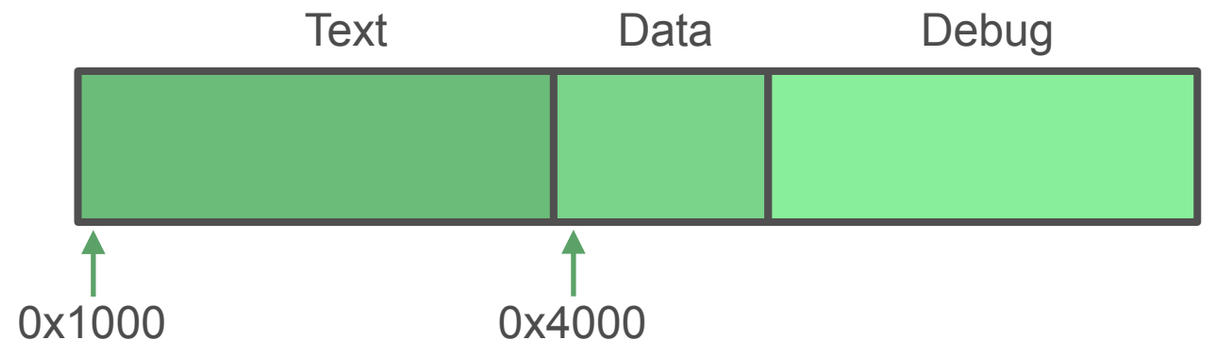
```
int var1 = 1;
int *var2 = &var1;

int function1()
{
  int i;
  i = function2();
  i += var1;
  return function3(i);
}

int function2()
{
  return *var2;
}
```

b.c

```
int function3(int x)
{
  return x * 5;
}
```



Symbol Table

Symbol	Address
function1	0x1000
function2	0x1100
var1	0x4000
var2	0x4008
function3	0x1200

Relocations

File	Section	Offset	Symbol
a.o	Text	0x38	function2
a.o	Text	0x44	var1
a.o	Text	0x54	function3
a.o	Text	0x110	var2
a.o	Data	0x8	var1

COMDAT Groups

a.h

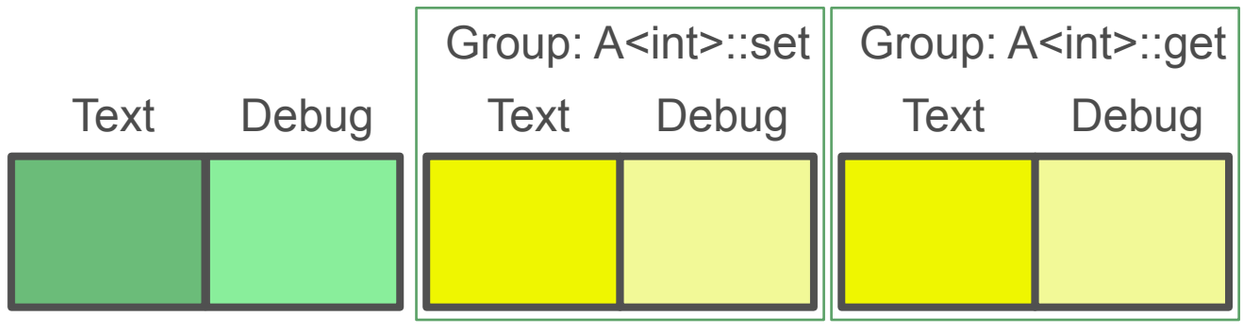
```
template <typename T>
class A {
 public:
  void set(T a);
  T get();
 private:
  T a_;
};

template <typename T>
void A<T>::set(T a)
{ a_ = a; }

template <typename T>
T A<T>::get()
{ return a_; }
```

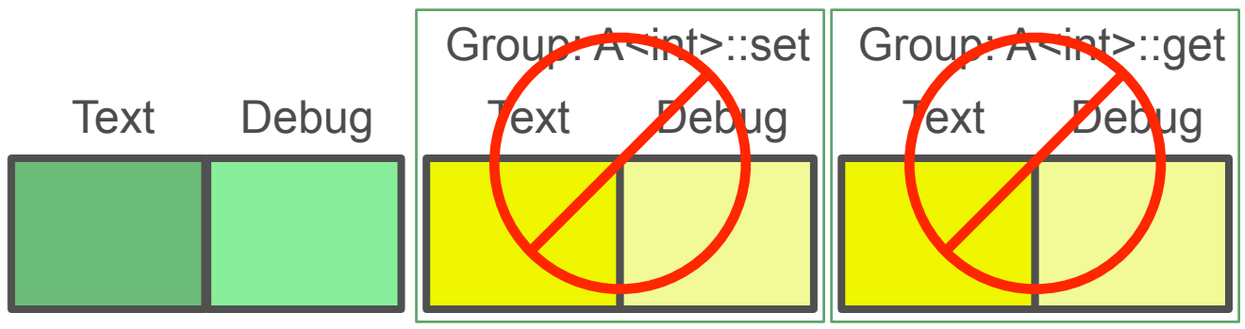
a1.cc

```
#include "a.h"
…
```



a2.cc

```
#include "a.h"
…
```



Dynamic Linking and Position-Independent Code

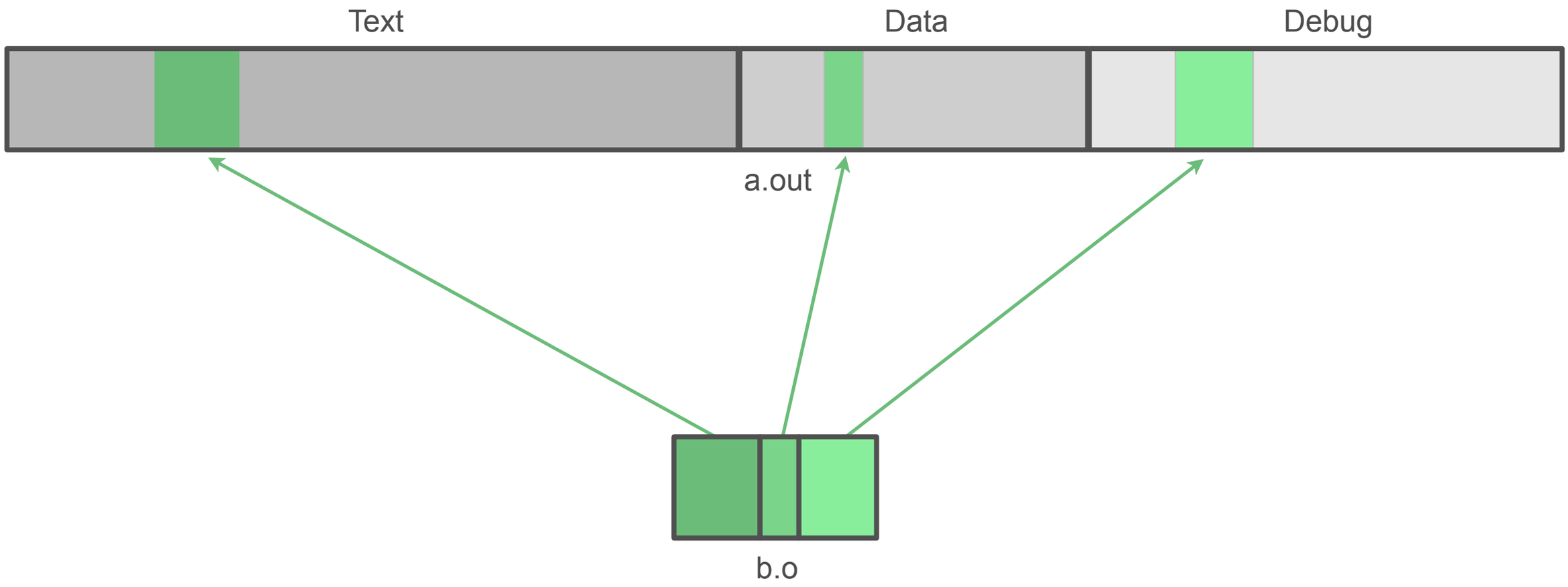
Global Offset Table (GOT)

- When compiling for a shared library, the -fPIC option generates position-independent code
- Each load module has a GOT, containing addresses of global variables and functions used in that module
- Loads and stores fetch a variable's address from the GOT

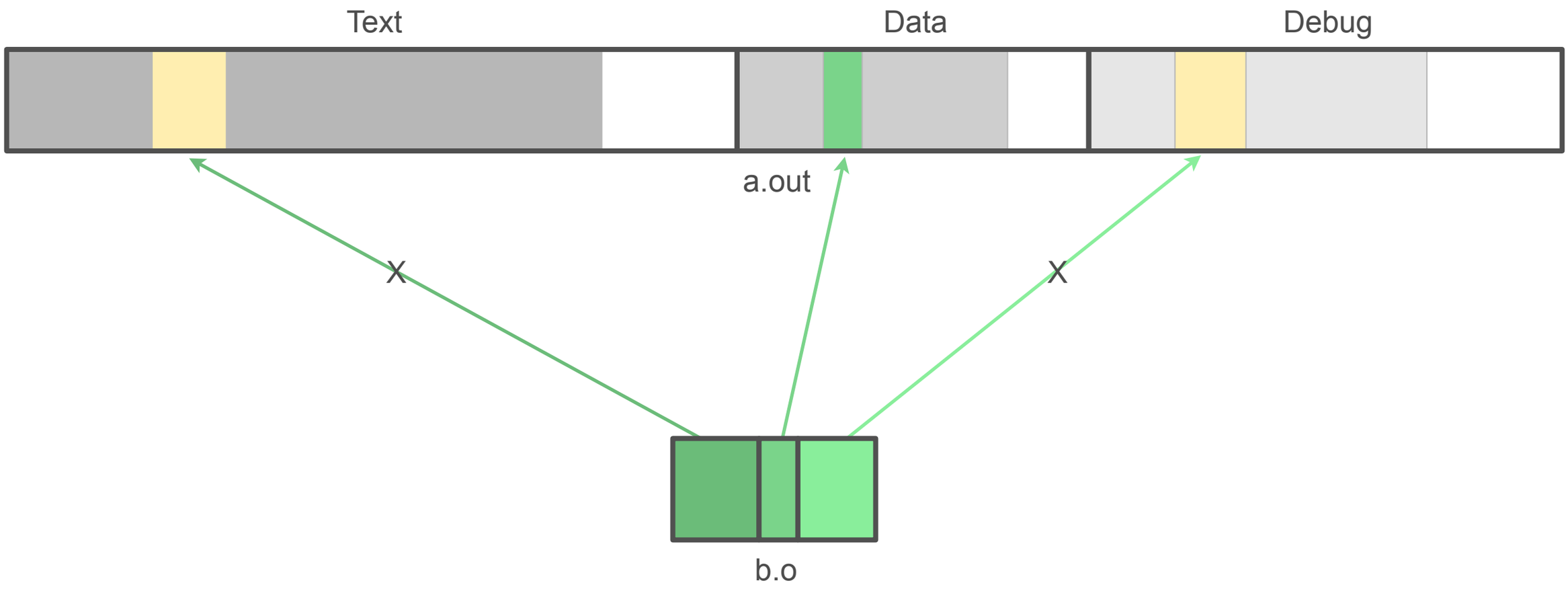
Procedure Linkage Table (PLT)

- Each load module has a PLT, containing a short code sequence for each external function called from that module
- Each PLT slot loads the function's address from the GOT, then branches to that address
- With lazy binding, the GOT entry initially points to a dynamic loader entry point

Incremental Update

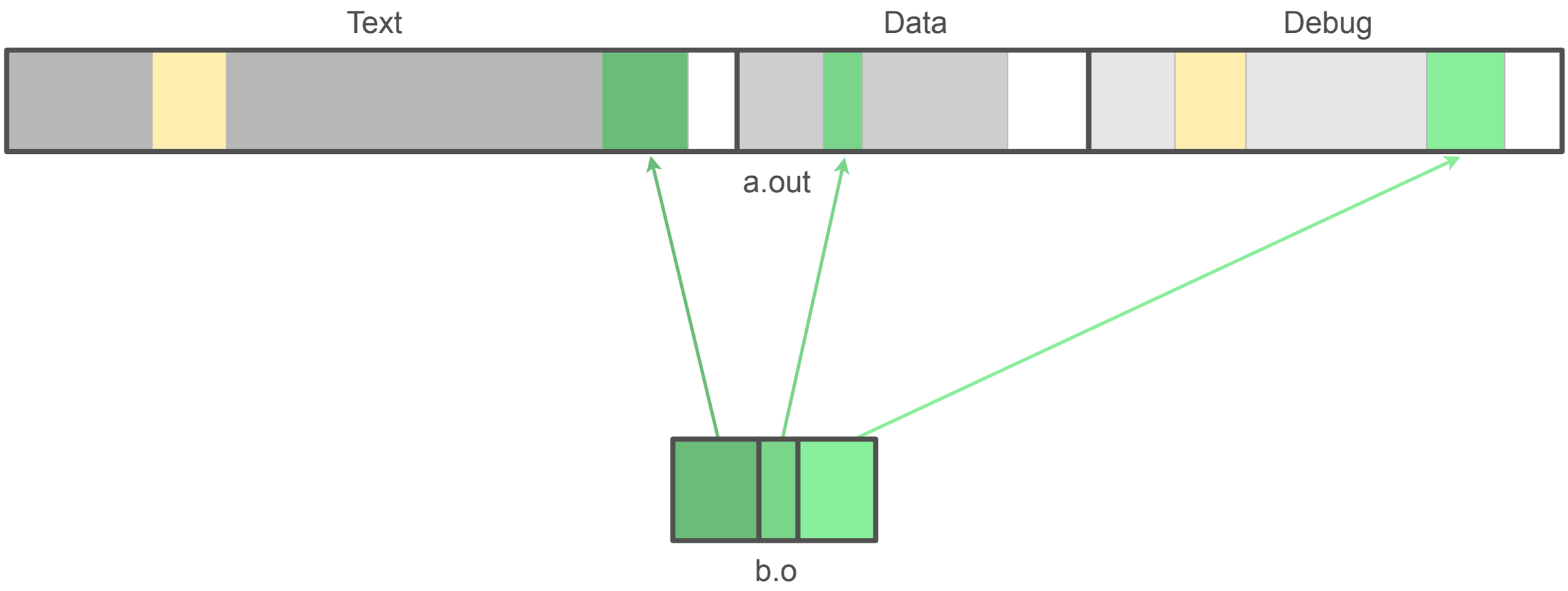


When New Code Doesn’t Fit



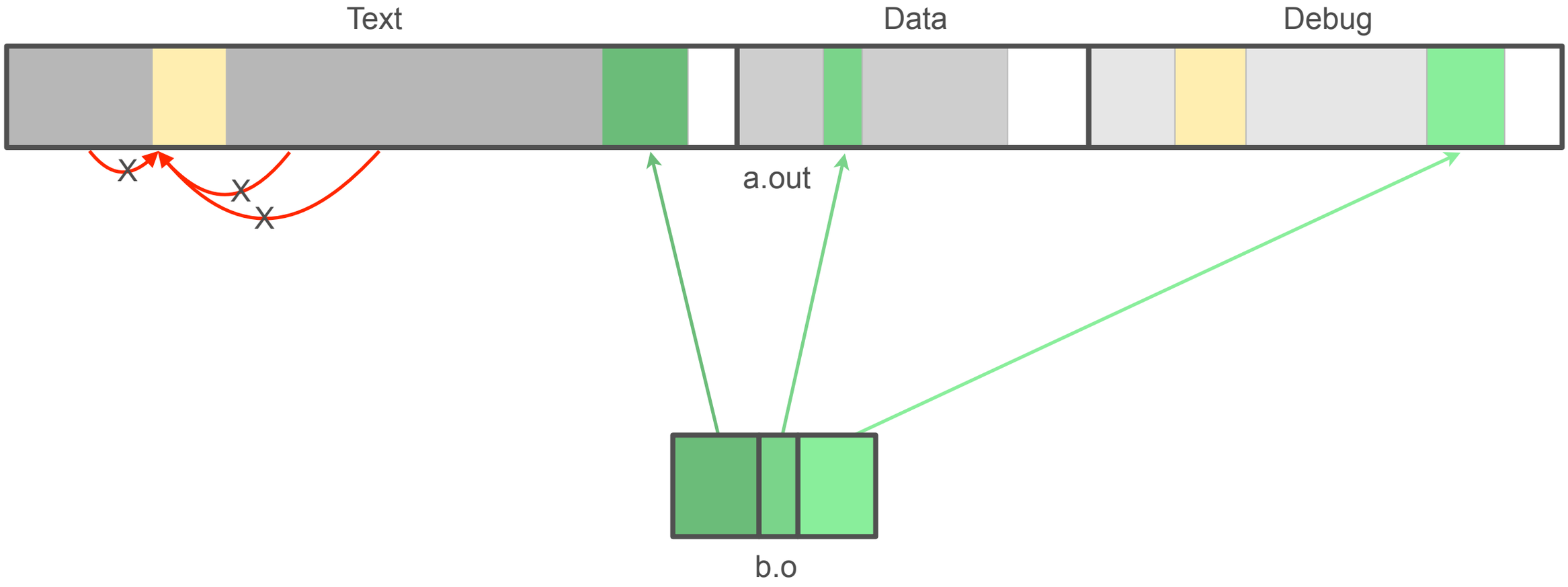
When New Code Doesn’t Fit

- Place code in pre-allocated patch space



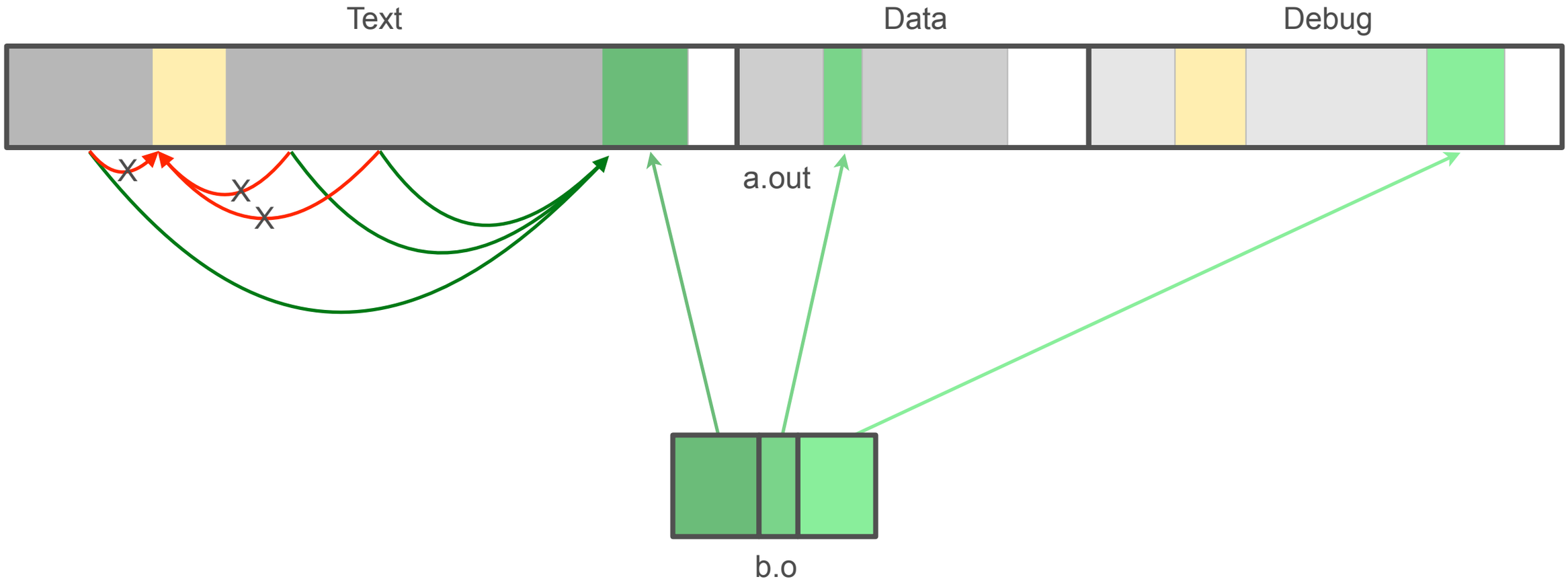
When New Code Doesn’t Fit

- Place code in pre-allocated patch space



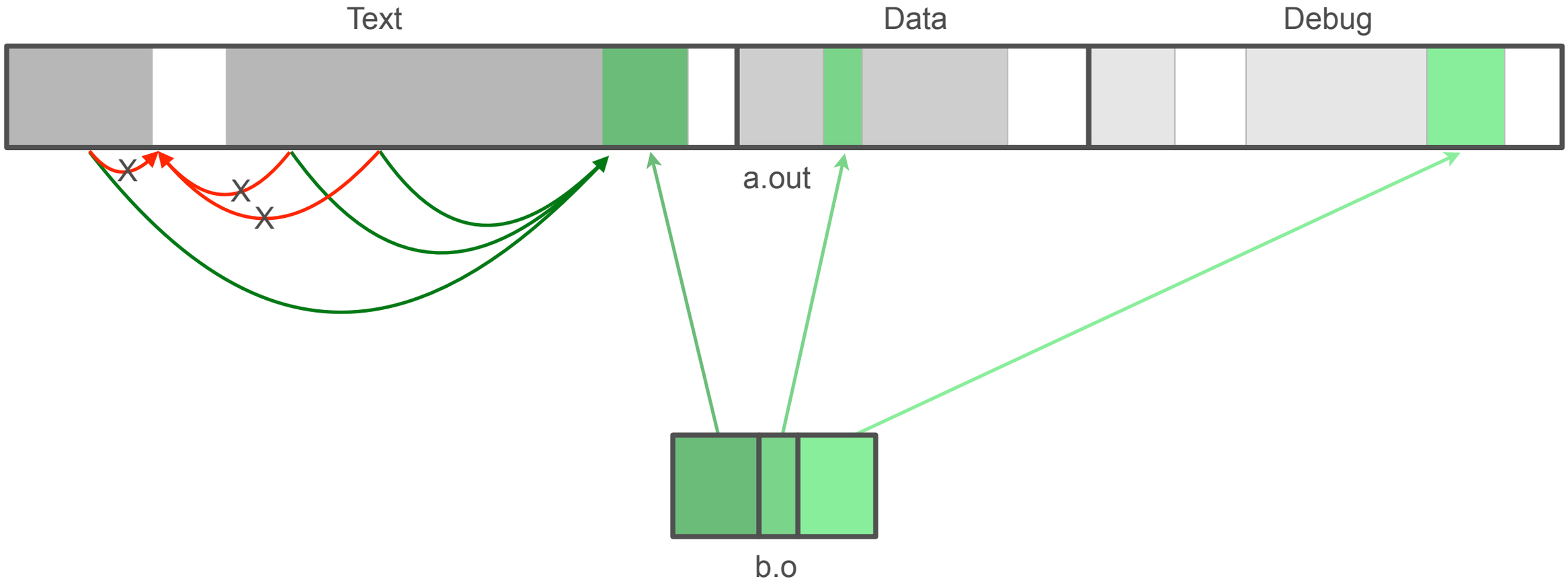
When New Code Doesn’t Fit

- Place code in pre-allocated patch space
- Relocate all references to new code



When New Code Doesn’t Fit

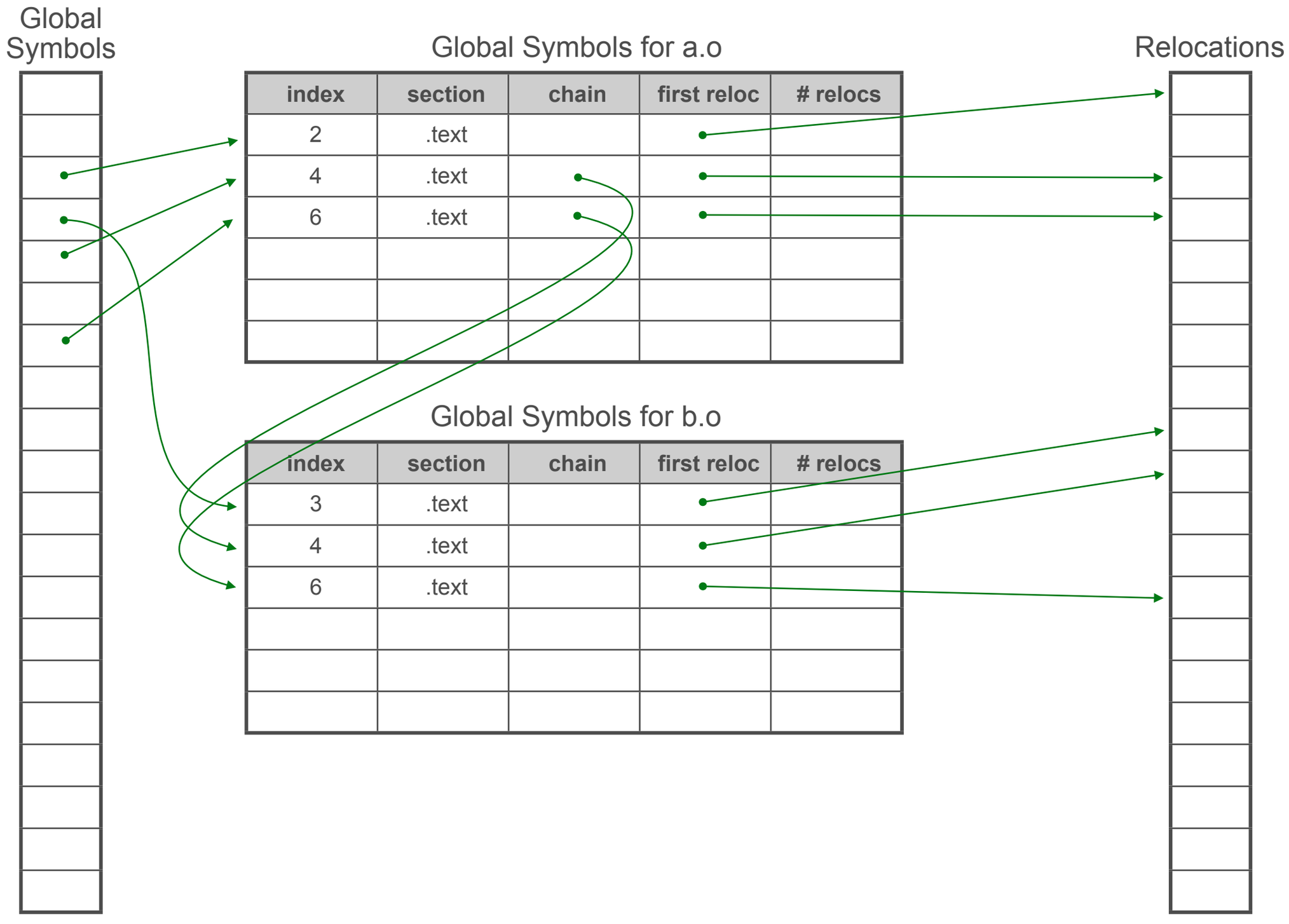
- Place code in pre-allocated patch space
- Relocate all references to new code



Bookkeeping

- Command line arguments
- Input files and timestamps
- Global symbols and relocations
- Sections
- COMDAT groups
- GOT/PLT entries
- Unused symbols from archive libraries

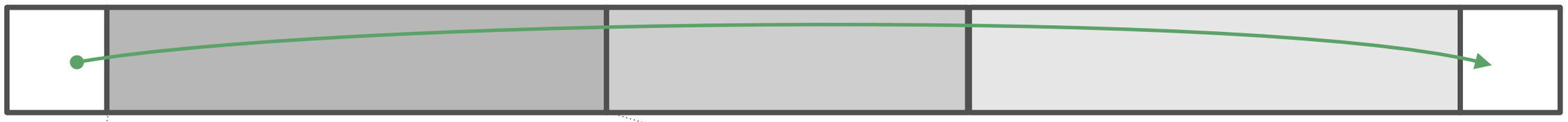
Incremental Symbols & Relocations



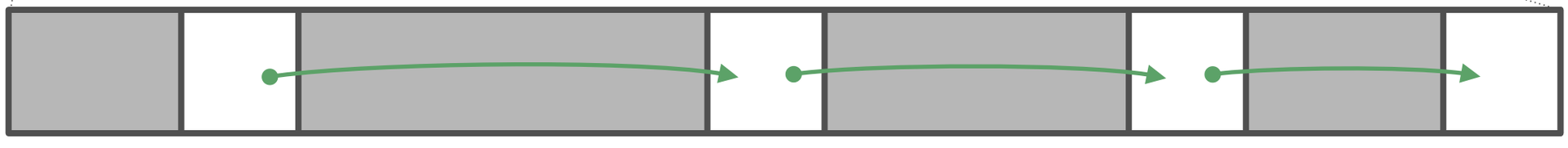
Free Space

- Maintain lists of free blocks at two levels
- Simple linked-list structure
- Build lists at start of link, removing used blocks for unchanged inputs
- First-fit lookup

First level: List of free blocks in the file



Second level: List of free blocks in each section



Command-Line Options

--incremental

- If no existing file, does a full link and prepares for incremental update
- Otherwise, checks timestamps to determine which files to update

--incremental-full

- Forces a full link and prepares for incremental update

--incremental-update

- Forces an incremental update
- Special exit code if update not possible

--incremental-patch=*n*

- Adds *n*% patch space to each section (default 10%)

Options for Distributed Build Systems

--incremental-base=filename

- Selects a file other than the output file as the base for an incremental update

--incremental-unchanged

- Subsequent files will be considered unchanged; no timestamp checks

--incremental-changed

- Subsequent files will be considered changed; no timestamp checks

--incremental-unknown

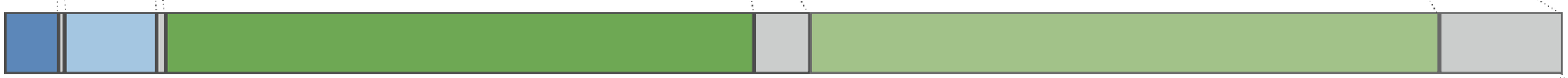
- Check timestamps on subsequent files

Overhead

In a typical executable, about 80% is debug info

Debug strings grow by about 10x because we don’t merge duplicates

Add 10% patch space to each section



Add 3–4% for incremental link bookkeeping

Limitations

- Intended for development use, not for building release binaries
- Not compatible with: --gc-sections, -r, --emit-relocs, --plugin
- Command line, including input files, must remain unchanged (except for certain options)
- Linker scripts must remain unchanged
- There must be enough patch space in each section
- No SHF_MERGE section processing
- Debug info sections require special handling
- No .eh_frame_hdr section (yet)
- Always rebuild: symbol tables, section table, program header table, dynamic table, dynamic relocations, and incremental info sections

Future

- Support .eh_frame_hdr and rebuild for incremental updates
- SHF_MERGE sections
- Automatic fallback to full link when out of patch space
- Incremental update of incremental info

Thank You