Transition to C

Review

- C0, C1

File simple.c0

```
#use <util>

/*** Interface ***/
int absval(int x)
/*@requires x > int_min(); @*/
/*@ensures \result >= 0; @*/ ;

struct point2d {
  int x;
  int y;
};

/*** Implementation ***/
int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  return x < 0 ? -x : x;
}
```

A simple library

File test.c0

```
#use <conio>

int main() {
  struct point2d* P = alloc(struct point2d);
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); print("\n");
  print("y coord: "); printint(P->y); print("\n");
  return 0;
}
```

A sample application that uses it

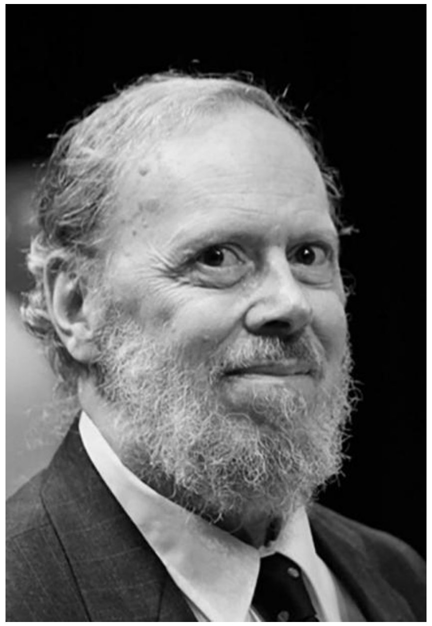
How we compile and run them

Linux Terminal

```
# cc0 -d simple.c0 test.c0
# ./a.out
x coord: -15
y coord: 30
0
```

The C Language

C



Dennis Ritchie

- C was designed in 1972 to implement Unix
  - People didn’t know how to design good languages back then
  - C is a terrible language
- C is pervasively used for system-level programming
  - C is very fast
- C0/C1 is a **safe** subset of C with **contract** annotations

Preempts many pitfalls of C

Teaches good programming practices

- C is much more powerful than C0/C1
  - with great powers come great responsibilities
  - and a lot dangers

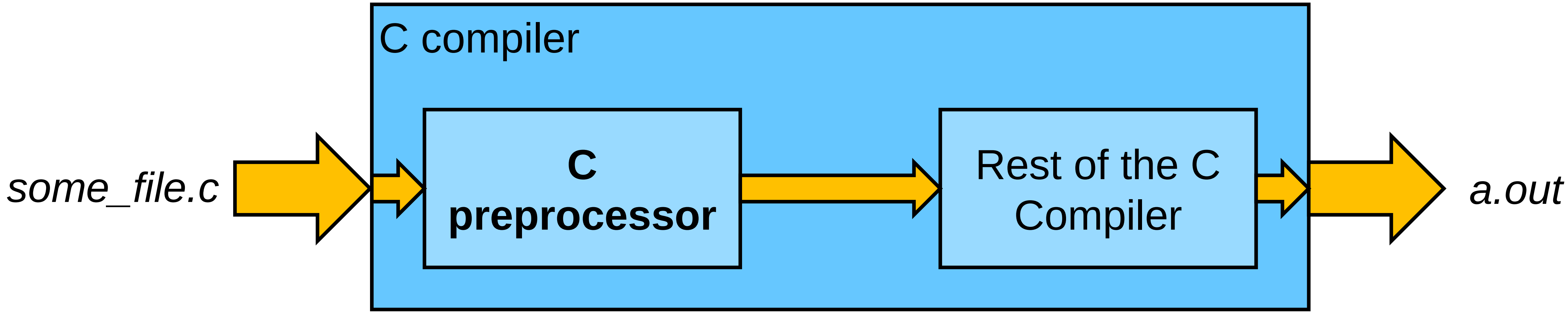
C0 **C**

The C Preprocessor

The C Preprocessor Language

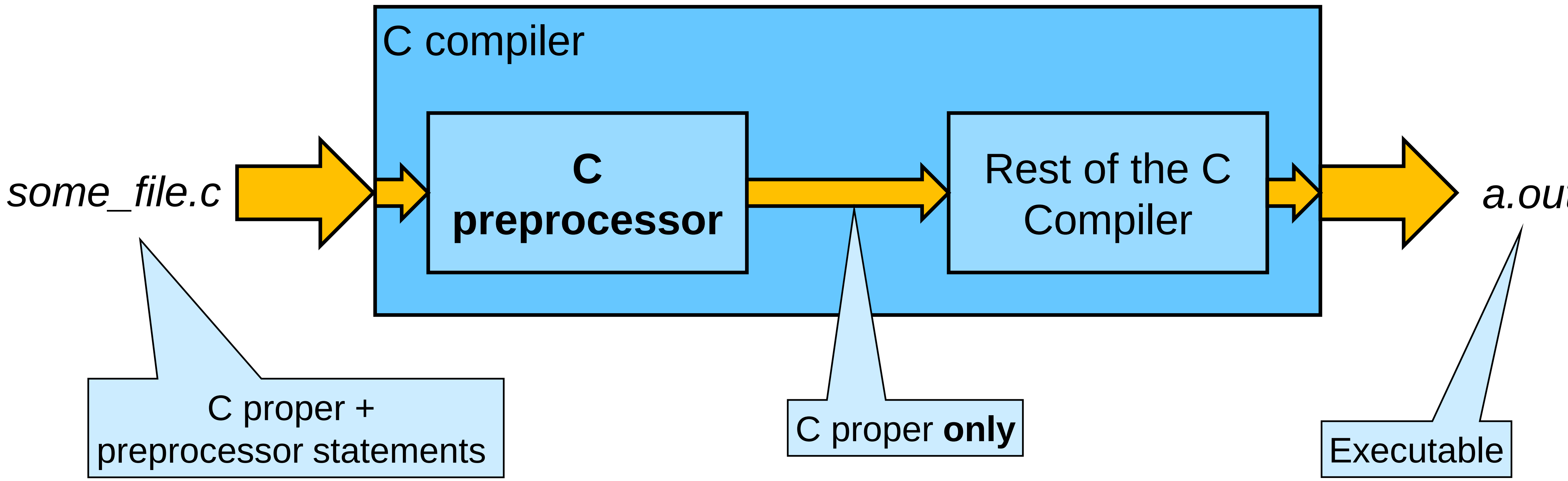
- A typical C program consists of statements written in **two** languages
  - the C preprocessor language
    - these *directives* all start with #
  - C proper

- The first thing the C compiler does is to call the **C preprocessor**
  - a program that processes all the C preprocessor directives
  - and produces code that is entirely in C proper



The C Preprocessor

- The C compiler first calls the **C preprocessor**



- This happens behind the scenes when compiling a C program
  - but the C preprocessor can also be run independently as the Unix command **cpp**

You won't need this

Useful C Preprocessor Directives

- File inclusion
- Macro definitions
- Conditional compilation
- Macro functions

There are many many more …

… but this is all we'll need

File Inclusion

- Used to dump the contents of a file in the current program
  - similar to C0's **#use** directive
    - but not exactly

More on this later

#include <stdio.h>

This is akin to C0's **#use** <conio>

- includes **system file** stdio.h in the current program

#include "lib/xalloc.h"

- includes **local file** lib/xalloc.h in the current program

We never had a need for this in C0

Header Files

- The only thing we **#include** in a C file is a header file
  - by convention they end in .h
    - e.g., stdio.h
- A **header file** contains
  - a library's interface
    - function prototypes
    - type definitions
  - other preprocessor directives
- Nothing prevents including other types of files
  - but it's rarely a good idea

Never #include a .c file

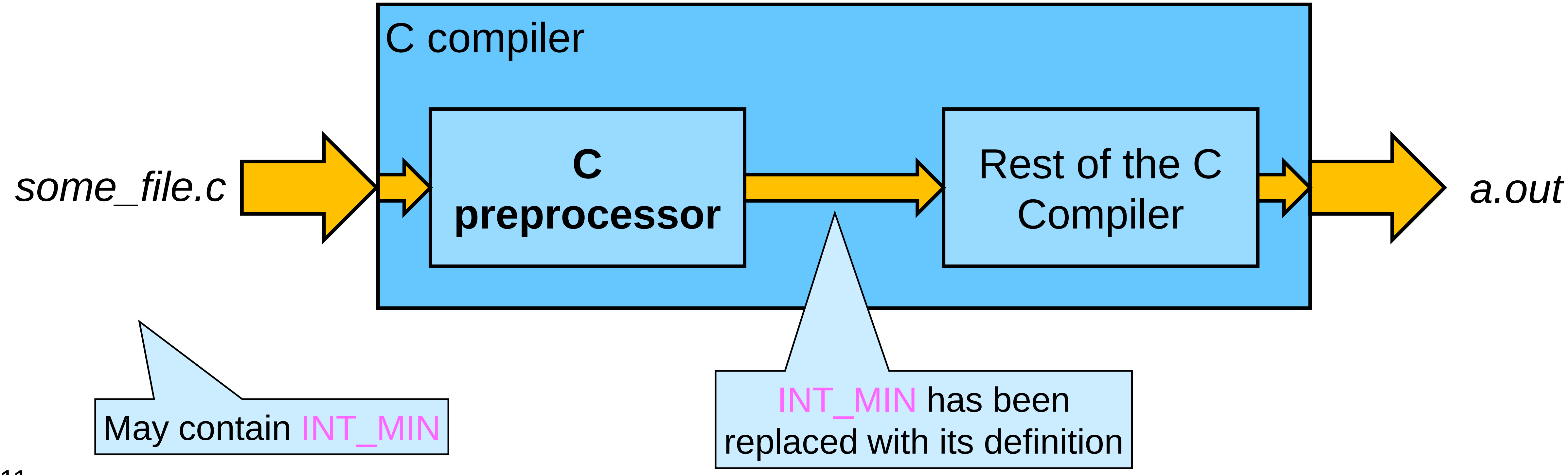
An endless source of bugs

Macro Definitions

- A way to give a name to a constant
    - Example

      #define INT_MIN 0x80000000

- The program can then use the **macro symbol** INT_MIN
- The preprocessor replaces every occurrence of INT_MIN with 0x80000000



Macro Definitions

- A way to give a name to a constant

A macro

```
#define INT_MIN 0x80000000
```

A (macro) symbol

its definition

- By convention, macro symbols are written in ALL CAPS

- This is a convenient way to give names to constants
  - like INT_MIN, the smallest value of type int

In C0, int_min() had to be a function because only types and function can be defined at the top level

Macro Definitions

- Macros definitions can be any expression
  - not just constants

Not the most obvious definition of INT_MAX, but bear with us

```
#define INT_MAX INT_MIN ^ -1
```

  - Then, the preprocessor will expand

    INT_MAX / 2

    to

    0x80000000 ^ -1 / 2

    which C understands as

    0x80000000 ^ (-1 / 2)

    that is **not** what we meant ✘

- The C preprocessor does not understand operator precedence

Macro Definitions

- The C preprocessor does not understand precedences
  - Add parentheses to communicate intention

```
#define INT_MAX (INT_MIN ^ -1)
```

  - Now, the preprocessor will expand

    INT_MAX / 2

    to

    (0x80000000 ^ -1) / 2

    which *is* what we meant

- **Use macro definitions with care**
  - You will not need to define any macro in this course
    - but you will need to know what they do

Conditional Compilation

- Allows compiling (or not) a program segment depending on whether a symbol is defined
  - Example

```
#ifdef DEBUG
  printf("Reached this point\n");
#endif
```

    - If the symbol DEBUG has been defined
      printf("Reached this point\n");
      will be compiled as part of the program
    - otherwise, it is cut out and **never reaches the compiler** proper

  - DEBUG can be defined with
    - **#define** DEBUG
    - or on the compilation command

No need to define it as anything

More on this later

Conditional Compilation

- We can provide an **#else** clause
  - Example

```
#ifdef X86_ARCH
  #include "arch/x86_optimizations.h"
  x86_optimize(code);
#else
  generic_optimize(code);
#endif
```

- We can also test if a symbol is **not** defined
  - Example

```
#ifndef INT_MIN
  #define INT_MIN 0x80000000
#endif
```

If INT_MIN is not defined, define it

Macro Function Definitions

- We can define macros with arguments
  - Example

```
#define MULT(x, y) x * y
```

  - Then, the preprocessor will expand

```
MULT(1 + 2, 3 - 5) / 2
```

    to

```
1 + 2 * 3 - 5 / 2
```

    which is **not** what we meant

The C preprocessor does not understand operator precedence

  - We need to add lots of parentheses

```
#define MULT(x, y) ((x) * (y))
```

- **Use macro function definitions with extreme care**
  - You will not need to define any macro in this course
    - but we will use exactly 3 of them

Contracts Emulation

- **C does not have contracts**
  - this means you are on your own when code doesn't work

- Some C0 contracts can be emulated

- The header file **contracts.h** provides the macro functions
  - REQUIRES(condition)
  - ENSURES(condition)
  - ASSERT(condition)

They serve the same purposes as //@requires, //@ensures and //@assert

//@loop_invariant can be emulated through judicious uses of ASSERT

Contracts Emulation

- The header file **contracts.h** provides the macro functions
  - REQUIRES(condition)
  - ENSURES(condition)
  - ASSERT(condition)

File lib/contract.h

```
#include <assert.h>

#ifdef ASSERT
#undef ASSERT
#endif

#ifdef REQUIRES
#undef REQUIRES
#endif

#ifdef ENSURES
#undef ENSURES
#endif

#ifndef DEBUG

#define ASSERT(COND) ((void)0)
#define REQUIRES(COND) ((void)0)
#define ENSURES(COND) ((void)0)

#else

#define ASSERT(COND) assert(COND)
#define REQUIRES(COND) assert(COND)
#define ENSURES(COND) assert(COND)

#endif
```

Do nothing

Undefine ASSERT were it to already have been defined

If we are not in DEBUG mode, define ASSERT to do nothing

Otherwise, define it as assert

Declares assert

Same thing for REQUIRES and ENSURES

Translating a C0 Program to C – I

High-level Approach

- We translate each file separately
  - test.c0 → test.c

- The library **interface** becomes a **header file**
  - simple.c0 → simple.h, simple.c

Translating a Library to C

File simple.c0

```c
#use <util>

/*** Interface ***/
int absval(int x)
/*@requires x > int_min(); @*/
/*@ensures \result >= 0; @*/ ;

struct point2d {
  int x;
  int y;
};

/*** Implementation ***/
int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  return x < 0 ? -x : x;
}
```

Interface

File simple.h

```c
/*** Interface ***/

int absval(int x)
/*@requires x > int_min(); @*/
/*@ensures \result >= 0;   @*/ ;

struct point2d {
  int x;
  int y;
};
```

Implementation

File simple.c

```c
/*** Implementation ***/
int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  return x < 0 ? -x : x;
}
```

We are not done translating this code

Translating a Library Interface to C

Interface

File simple.h

```
/*** Interface ***/

int absval(int x)
/*@requires x > int_min(); @*/
/*@ensures \result >= 0;   @*/ ;

struct point2d {
  int x;
  int y;
};
```

This is valid C code already

Prototype contracts are comments in C

//@ …; and /*@ …; @*/ do not have special meaning

We will need to update this header file slightly later

Implementation

Translating a Library to C

- Translating the **implementation**, line by line

File simple.c

```
/*** Implementation ***/
int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  return x < 0 ? -x : x;
}
```

- This is valid C up to here
- absval is mentioned in the header file simple.h
  - we need to #include it

Translating a Library to C

- Translating the **implementation**, line by line

File simple.c

```
/*** Implementation ***/
#include "simple.h"

int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  return x < 0 ? -x : x;
}
```

- Next we need to translate the precondition
  - //@requires becomes REQUIRES in the body of the function
  - for this, we have to #include contracts.h

We keep it in local directory lib/

Translating a Library to C

- Translating the **implementation**, line by line

File simple.c

```
/*** Implementation ***/
#include "simple.h"
#include "lib/contracts.h"

int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  REQUIRES(x > int_min());
  return x < 0 ? -x : x;
}
```

This is now a comment

- int_min() is not predefined in C
- but the symbol INT_MIN represents the smallest integer
  - it is defined in the system header file <limits.h>

Translating a Library to C

- Translating the **implementation**, line by line

File simple.c

```
/*** Implementation ***/
#include "simple.h"
#include "lib/contracts.h"
#include <limits.h>

int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  REQUIRES(x > INT_MIN);
  return x < 0 ? -x : x;
}
```

- Next is the postcondition
  - //@ensures becomes ENSURES **before every return statement**
    - in general, every place the function may return
  - because we are returning a complex expression, we need to save it in a temporary variable
    - call it result since it contains the value of \result

Translating a Library to C

- Translating the **implementation**, line by line

File simple.c

```c
/*** Implementation ***/
#include "simple.h"
#include "lib/contracts.h"
#include <limits.h>

int absval(int x)
//@requires x > int_min();
//@ensures \result >= 0;
{
  REQUIRES(x > INT_MIN);
  int result = x < 0 ? -x : x;
  ENSURES(result >= 0);
  return result;
}
```

- All remaining code
  - either was added during the translation
  - or was valid C already
- We are done

Translating a C0 Program to C – II

Translating a Program File to C

- We now translate the client of the library

File test.c0

```
#use <conio>

int main() {
  struct point2d* P = alloc(struct point2d);
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); print("\n");
  print("y coord: "); printint(P->y); print("\n");
  return 0;
}
```

File test.c

```
#use <conio>

int main() {
  struct point2d* P = alloc(struct point2d);
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); print("\n");
  print("y coord: "); printint(P->y); print("\n");
  return 0;
}
```

We are not done translating this code

Translating a Program File to C

- Let's proceed again line by line

File test.c

```
#use <conio>

int main() {
  struct point2d* P = alloc(struct point2d);
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); print("\n");
  print("y coord: "); printint(P->y); print("\n");
 return 0;
}
```

  - The input/ouput system functions of C are declared in header file <stdio.h>

Translating a Program File to C

- Let's proceed again line by line

File test.c

```
#include <stdio.h>

int main() {
  struct point2d* P = alloc(struct point2d);
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); print("\n");
  print("y coord: "); printint(P->y); print("\n");
  return 0;
}
```

- The function header is valid C
- The way allocated memory is appropriated is different in C
  - this is done by calling malloc
- malloc takes a **size**
  - the number of bytes to allocate
    - in C0, alloc took a *type*
  - the function sizeof computes the number of bytes a **type** takes up in memory
  - both malloc and sizeof are defined in <stdlib.h>

Translating a Program File to C

- Let's proceed again line by line

File test.c

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

int main() {
  struct point2d* P = malloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); print("\n");
  print("y coord: "); printint(P->y); print("\n");
  return 0;
}
```

- malloc returns NULL when there isn't enough memory
  - the next dereference will be unsafe
    - and be really hard to debug
  - A better approach is to abort
- The library xalloc.h defines xmalloc
  - which **fails** if there is not enough memory
  - better use that

We keep it in local directory lib/

Translating a Program File to C

- Let’s proceed again line by line

File test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); print("\n");
  print("y coord: "); printint(P->y); print("\n");
  return 0;
}
```

  - assert is defined in system header file <assert.h>

Translating a Program File to C

- Let's proceed again line by line

File test.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  print("x coord: "); printint(P->x); print("\n");
  print("y coord: "); printint(P->y); print("\n");
  return 0;
}
```

- Printing in C is done using the function printf defined in <stdio.h>
- printf takes a **format string** with
  - placeholders for the values to print
  - and these values as additional arguments

printf takes a *variable number* of arguments

Translating a Program File to C

- Let's proceed again line by line

File test.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

%d means print the argument as a decimal number

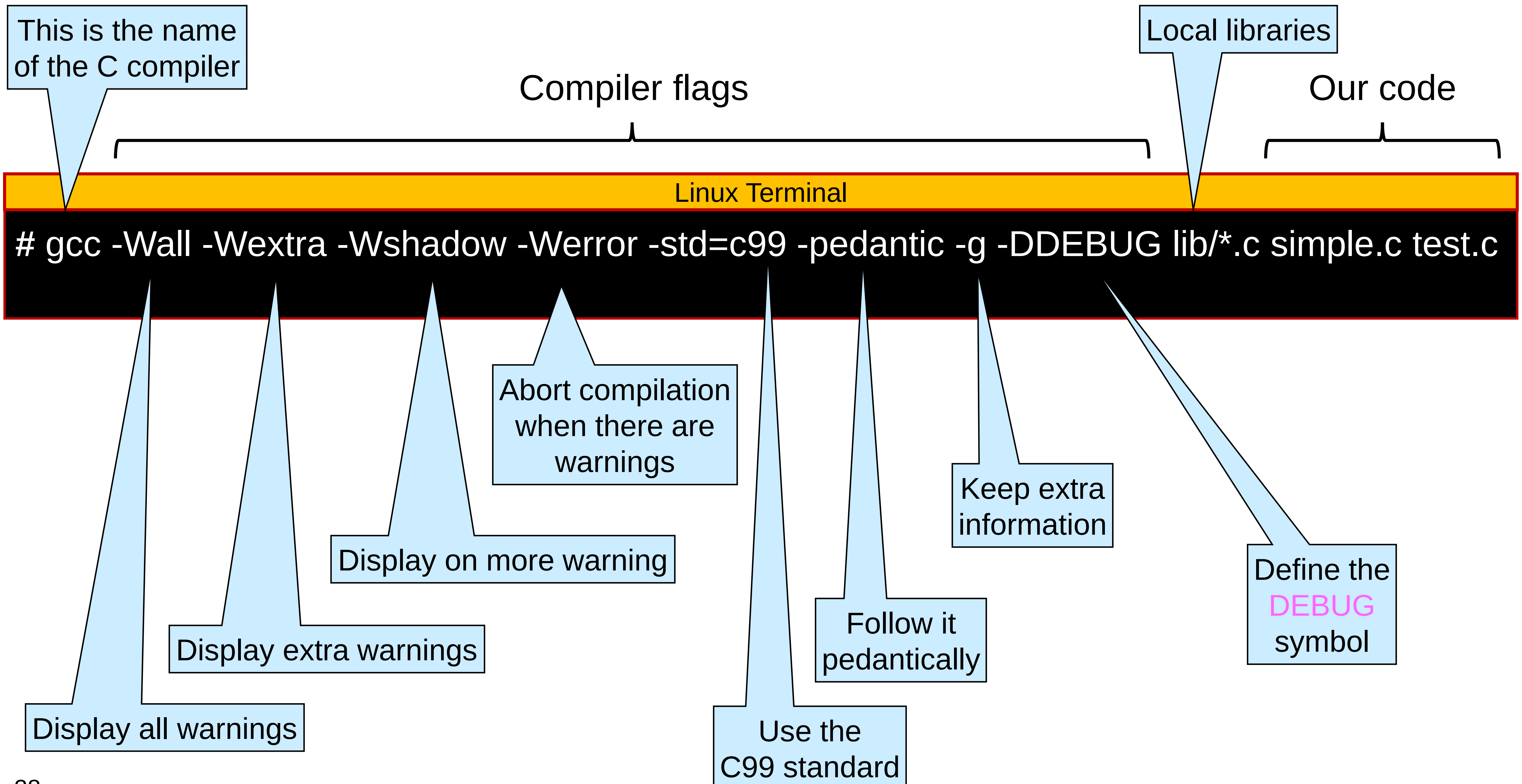
There are lots of different placeholders

- At this point, we have C code we can compile

Compiling a C Program

Compiling a C Program

- Here’s how to compile our translated example



Compiling a C Program

- Here's how to compile our translated example

Local libraries

Our code

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
```

  - Notice that we **only compile .c files**
    - not header file

- *Let's do it!*

Compiling Our Program

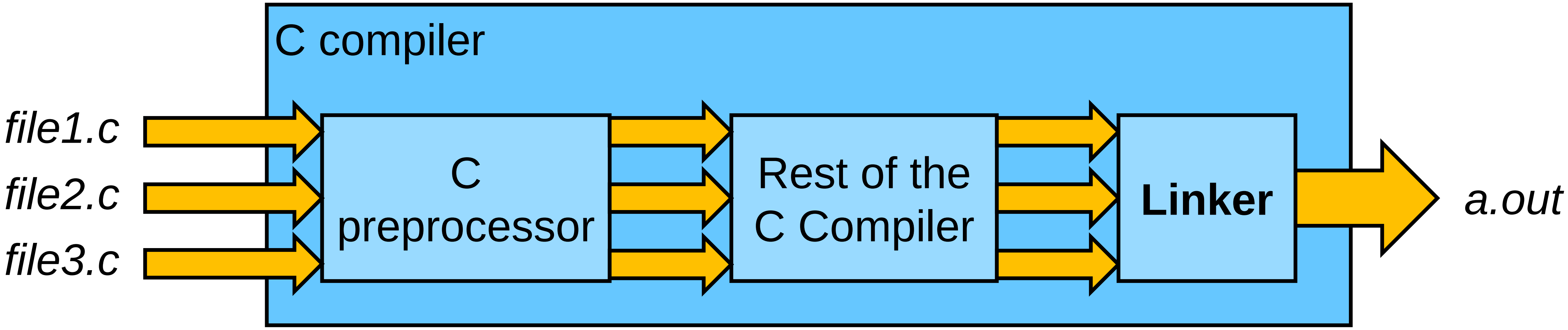
Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
test.c: In function ‘main’:
test.c:6:38: error: invalid application of ‘sizeof’ to incomplete type ‘struct point2d’
   struct point2d* P = xmalloc(sizeof(struct point2d));
                                      ^~~~~~
test.c:7:4: error: dereferencing pointer to incomplete type ‘struct point2d’
   P->x = -15;
    ^~
test.c:8:17: error: implicit declaration of function ‘absval’; did you mean ‘abs’? [-
   Werror=implicit-function-declaration]
   P->y = P->y + absval(P->x * 2);
                 ^~~~~~
…
```

- Lots of errors!
  - These three are about struct point2d and absval
  - gcc does not know about these names when compiling test.c

Separate Compilation

- When compiling multiple file in C0, they are combined in single file that gets compiled
- In C, each file is compiled **separately**
  - then the compiled files are combined into a.out by the **linker**



  - Each file needs to know about all the names it uses
    - #include the header files containing those names

Including Header Files

- #include simple.h to provides the missing names to test.c

File test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>
#include "simple.h"

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

Including Header Files

- Before we compile again …

File test.0

```
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>
#include "simple.h"
#include "simple.h"

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

- A header file can end up included multiple times
  - often via other header files
- Let’s see what happens if we #include simple.h twice

Compiling Our Program

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
In file included from test.c:5:0:
simple.h:10:8: error: redefinition of 'struct point2d'
 struct point2d {
        ^~~~~~~
In file included from test.c:4:0:
simple.h:10:8: note: originally defined here
 struct point2d {
        ^~~~~~~
…
```

- struct point2d is defined twice
  - once each time we **#include** simple.h
- This is an error

C0 notices this and skips **#use**s beyond the first one

Hardening a Library Interface

- Use conditional compilation to makes sure the definitions in a header file are included just once

_SIMPLE_H_ is some unique symbol

Interface

File simple.h

```
/*** Interface ***/

#ifndef _SIMPLE_H_
#define _SIMPLE_H_

int absval(int x)
/*@requires x > int_min(); @*/
/*@ensures \result >= 0;   @*/ ;

struct point2d {
  int x;
  int y;
};

#endif
```

- If _SIMPLE_H_ **is not** defined
  - define it
  - provide the interface definitions
- If _SIMPLE_H_ **is** defined
  - do nothing
- The first time we #include simple.h
  - _SIMPLE_H_ **is not** defined
    - the interface definitions are #included
- Any time after that
  - _SIMPLE_H_ **is** defined
    - the interface definitions are not #included

Compiling Our Program

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
test.c: In function ‘main’:
test.c:12:25: error: ‘true’ undeclared (first use in this function); did you mean ‘free’?
   assert(P->y > P->x && true);
                         ^
#
```

- One error only!
  - true?

- bool is not a primitive type in C
  - To use the booleans, we need to #include <stdbool.h>

Booleans

- bool is not a primitive type in C
  - To use booleans, we need to #include <stdbool.h>

File test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>
#include "simple.h"
#include "simple.h"
#include <stdbool.h>

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

Compiling Our Program

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
#
```

- Success!
- Let’s run it

Translating a C0 Program to C – III

Running Our Program

```c
#include …

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

- Let’s run it

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
# ./a.out
x coord : -15
y coord: 1073741854
```

- 1073741854 does not look right
  - C0 gave us back 30
- **C does not initialize allocated memory to default value**
  - it uses whatever is at that location
- Let’s run it again

Running Our Program

```c
#include …

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

- Let's run it again
  - C does not initialize allocated memory to default value

Linux Terminal

```
# ./a.out
x coord : -15
y coord: 1073741854
# ./a.out
Assertion failed: (P->y > P->x && true, function main, file test.c, line 13.
Abort trap: 6
# ./a.out
x coord : -15
y coord: 30
# ./a.out
x coord : -15
y coord: 1879048222
```

- Different executions produce different values
  - This is an endless source of bugs

This was impossible in C0

Running Our Program

- C does not initialize allocated memory to default value
  - This makes C **fast**
  - But this is **dangerous**

File test.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>
#include "simple.h"
#include "simple.h"
#include <stdbool.h>

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

- The obvious fix is to initialize P->y
- But it is rarely this obvious
- A more systematic way to find uninitialized memory bugs (and others) is to use the **valgrind** tool

Valgrind

- Just type valgrind in front of executable

Linux Terminal

```
# valgrind ./a.out
==9073== Memcheck, a memory error detector
==9073== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9073== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==9073== Command: a.out
==9073==
==9073== Conditional jump or move depends on uninitialised value(s)
==9073==    at 0x10891B: main (test.c:13)
==9073==
x coord: -15
…
```

Valgrind notices that a statement depending on an unitialized value has been executed

Approximate line where this occurred

Initializing Memory

- C does not initialize allocated memory to default value
  - Fix it by initializing P->y

File test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>
#include "simple.h"
#include "simple.h"
#include <stdbool.h>

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = 0;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

- Let’s try now again

Initializing Memory

- Let's recompile and run it again

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
# ./a.out
x coord: -15
y coord: 30
```

- This is the expected output
  - Same as with C0

Initializing Memory

- Let's run it with valgrind too

Linux Terminal

```
# valgrind ./a.out
…
x coord: -15
y coord: 30
==9197==
==9197== HEAP SUMMARY:
==9197==     in use at exit: 8 bytes in 1 blocks
==9197==   total heap usage: 2 allocs, 1 frees, 1,032 bytes allocated
==9197==
==9197== LEAK SUMMARY:
==9197==    definitely lost: 8 bytes in 1 blocks
==9197==    indirectly lost: 0 bytes in 0 blocks
==9197==      possibly lost: 0 bytes in 0 blocks
==9197==    still reachable: 0 bytes in 0 blocks
==9197==         suppressed: 0 bytes in 0 blocks
==9197== Rerun with --leak-check=full to see details of leaked memory
…
```

Memory Leaks

- When the program exits, 8 bytes are still in use
  - that's the struct point2d it allocated
- C0 and C manage memory differently
  - C0 is *garbage-collected*
    - memory is reclaimed whenever needed
  - In C, the programmer needs to **free** allocated memory once it is not used any more
    - memory is never reclaimed
- A program has a **memory leak** if unused memory is not freed
  - in long-running programs
    - games, browsers, operating systems, …

  memory leaks cause the program to get slower and slower and eventually crash

Memory Leaks

- A program has a **memory leak** if unused memory is not freed
  - We avoid this by **freeing** allocated memory once it is not used any more
  - By the end of a program, no allocated memory shall be still in use

- The C motto

If you allocate it, you free it

Freeing Memory

- In C, the programmer needs to **free** allocated memory once it is not used any more

File test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>
#include "simple.h"
#include "simple.h"
#include <stdbool.h>

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = 0;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  free(P);
  return 0;
}
```

- Let’s run valgrind again

Freeing Memory

- Let's run it with valgrind again

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
# valgrind ./a.out
…
x coord: -15
y coord: 30
==9519==
==9519== HEAP SUMMARY:
==9519==     in use at exit: 0 bytes in 0 blocks
==9519==   total heap usage: 2 allocs, 2 frees, 1,032 bytes allocated
==9519==
==9519== All heap blocks were freed -- no leaks are possible
…
```

Freeing Memory Wrong

- We must **not** free memory before we are done using it

File test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>
#include "simple.h"
#include "simple.h"
#include <stdbool.h>

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = 0;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  free(P);
  printf("y coord: %d\n", P->y);
  return 0;
}
```

- Let's run valgrind again

Freeing Memory Wrong

File test.c

```
…
free(P);
printf("y coord: %d\n", P->y);
…
```

- Let's run it with valgrind again

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
# valgrind ./a.out
…
x coord: -15
==9550== Invalid read of size 4
==9550==    at 0x1089B0: main (test.c:17)
==9550==  Address 0x522d044 is 4 bytes inside a block of size 8 free'd
==9550==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9550==    by 0x1089AB: main (test.c:16)
==9550==  Block was alloc'd at
==9550==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
   linux.so)
==9550==    by 0x1088D4: xmalloc (xalloc.c:29)
==9550==    by 0x10891D: main (test.c:10)
   …
```

Line where the bad access occurred

Line where that memory was freed

Line where it was allocated

Freeing Memory Wrong

- We must not free memory more than once

File test.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lib/xalloc.h"
#include <assert.h>
#include "simple.h"
#include "simple.h"
#include <stdbool.h>

int main() {
  struct point2d* P = xmalloc(sizeof(struct point2d));
  P->x = -15;
  P->y = 0;
  P->y = P->y + absval(P->x * 2);
  assert(P->y > P->x && true);
  printf("x coord: %d\n", P->x);
  printf("y coord: %d\n", P->y);
  free(P);
  free(P);
  return 0;
}
```

- Let's run valgrind again

Freeing Memory Wrong

File test.c

```
…
free(P);
free(P);
…
```

- Let's run it with valgrind again

Linux Terminal

```
# gcc -Wall -Wextra -Wshadow -Werror -std=c99 -pedantic -g -DDEBUG lib/*.c simple.c test.c
# valgrind ./a.out
…
x coord: -15
y coord: 30
==9631== Invalid free() / delete / delete[] / realloc()
==9631==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9631==    by 0x1089D1: main (test.c:18)
==9631==  Address 0x522d040 is 0 bytes inside a block of size 8 free'd
==9631==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==9631==    by 0x1089C5: main (test.c:17)
==9631==  Block was alloc'd at
==9631==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
   linux.so)
==9631==    by 0x1088D4: xmalloc (xalloc.c:29)
==9631==    by 0x10891D: main (test.c:10)
   …
```

Line where the memory was freed again

Line where it was freed the first time

Line where it was allocated

Memory Ownership

Data Structure Libraries in C

- Data structures allocate memory
  - e.g., a BST implementation of a dictionary
    - all the nodes of the BST
    - the dictionary header
  - This memory must be freed
- The interface must provide a function to free it

Client Interface

```
typedef  void* entry;
typedef  void* key;

typedef  key entry_key_fn(entry  e)
  /*@requires e != NULL;                      @*/ ;

typedef bool key_compare_fn(key k1, key k2)
  /*@ensures -1 <= \result && \result <= 1;  @*/ ;
```

Library Interface

```
// typedef ______* dict_t;

dict_t  dict_new(entry_key_fn* entry_key,
                 key_compare_fn* compare)
  /*@requires entry_key != NULL && compare != NULL   @*/
  /*@ensures \result != NULL;                        @*/ ;

entry dict_lookup(dict_t D, key k)
  /*@requires D != NULL;                             @*/ ;

void dict_insert(dict_t D, entry e)
  /*@requires D != NULL && e != NULL;                @*/ ;

entry dict_min(dict_t D)
  /*@requires D != NULL;                             @*/ ;

void dict_free(dict_t D)
  /*@requires D != NULL;                             @*/ ;
```

Data Structure Libraries in C

- Data structures allocate memory
  - This memory must be freed
    - all the nodes of the BST and the dictionary header

- But what about the data itself
    - e.g., the *entries* the client stored in the dictionary
  - The library should not always free them
    - because the client may need them later
  - But sometimes it should
    - because the client won't need them later
  - In any case, only the client knows how to free the data

- Give the client a choice of whether the library should free the data or not
  - specify who **owns** the data when freeing the data structure

Memory Ownership

- The library needs the client to specify who **owns** the memory used by the data
- The client can declare a function that frees the data
- dict_free takes such a function as a second argument
  - if called with an actual function, it will use it to free the data
  - if called with NULL, it will leave the data alone

Client Interface

```
typedef  void* entry;
typedef  void* key;

typedef  key entry_key_fn(entry  e)
  /*@requires e != NULL;                    @*/ ;

typedef bool key_compare_fn(key k1, key k2)
  /*@ensures -1 <= \result && \result <= 1;  @*/ ;

typedef void entry_free_fn(entry e);
```

Library Interface

```
// typedef ______* dict_t;

dict_t  dict_new(entry_key_fn* entry_key,
                 key_compare_fn* compare)
  /*@requires entry_key != NULL && compare != NULL  @*/
  /*@ensures \result != NULL;                       @*/ ;

entry dict_lookup(dict_t D, key k)
  /*@requires D != NULL;                            @*/ ;

void dict_insert(dict_t D, entry e)
  /*@requires D != NULL && e != NULL;               @*/ ;

entry dict_min(dict_t D)
  /*@requires D != NULL;                            @*/ ;

void dict_free(dict_t D, entry_free_fn* Fr)
  /*@requires D != NULL;                            @*/ ;
```

Memory Ownership

- Library implementation

```
/*** BST dictionary Implementation ***/

void tree_free(tree *T, entry_free_fn *Fr) {
  REQUIRES(is_bst(T));
  if (T == NULL) return;

  if (Fr != NULL) (*Fr)(T->data);
  tree_free(T->left, Fr);
  tree_free(T->right, Fr);
  free(T);
}

void dict_free(dict *D, entry_free_fn *Fr) {
  REQUIRES(is_dict(D));
  tree_free(D->root, Fr);
  free(D);
}
```

If Fr is not NULL,
it is used to free the data

Free each node of the tree

Free the dictionary header

Library Interface

```
// typedef ______* dict_t;

dict_t  dict_new(entry_key_fn* entry_key,
                 key_compare_fn* compare)
  /*@requires entry_key != NULL && compare != NULL  @*/
  /*@ensures \result != NULL;                        @*/ ;

entry dict_lookup(dict_t D, key k)
  /*@requires D != NULL;                             @*/ ;

void dict_insert(dict_t D, entry e)
  /*@requires D != NULL && e != NULL;                @*/ ;

entry dict_min(dict_t D)
  /*@requires D != NULL;                             @*/ ;

void dict_free(dict_t D, entry_free_fn* Fr)
  /*@requires D != NULL;                             @*/ ;
```

Summary

Balance Sheet

Lost	***Gained***
• Contracts • Safety • Garbage collection • Memory initialization	• Preprocessor • Whimsical execution • Explicit memory management • Separate compilation