LECTURE 9 · CSE 333 · Summer 2026

C++: pointer, references, const, auto

Discussion: What's the most difficult CSE assignment you've completed? Did you feel proud?

Reminders

Assessments:

- Exercise 4 will be due Friday at 11:59pm
  - Includes C and C++
- HW2 will be released tomorrow
  - Due next Thursday at 11:59pm

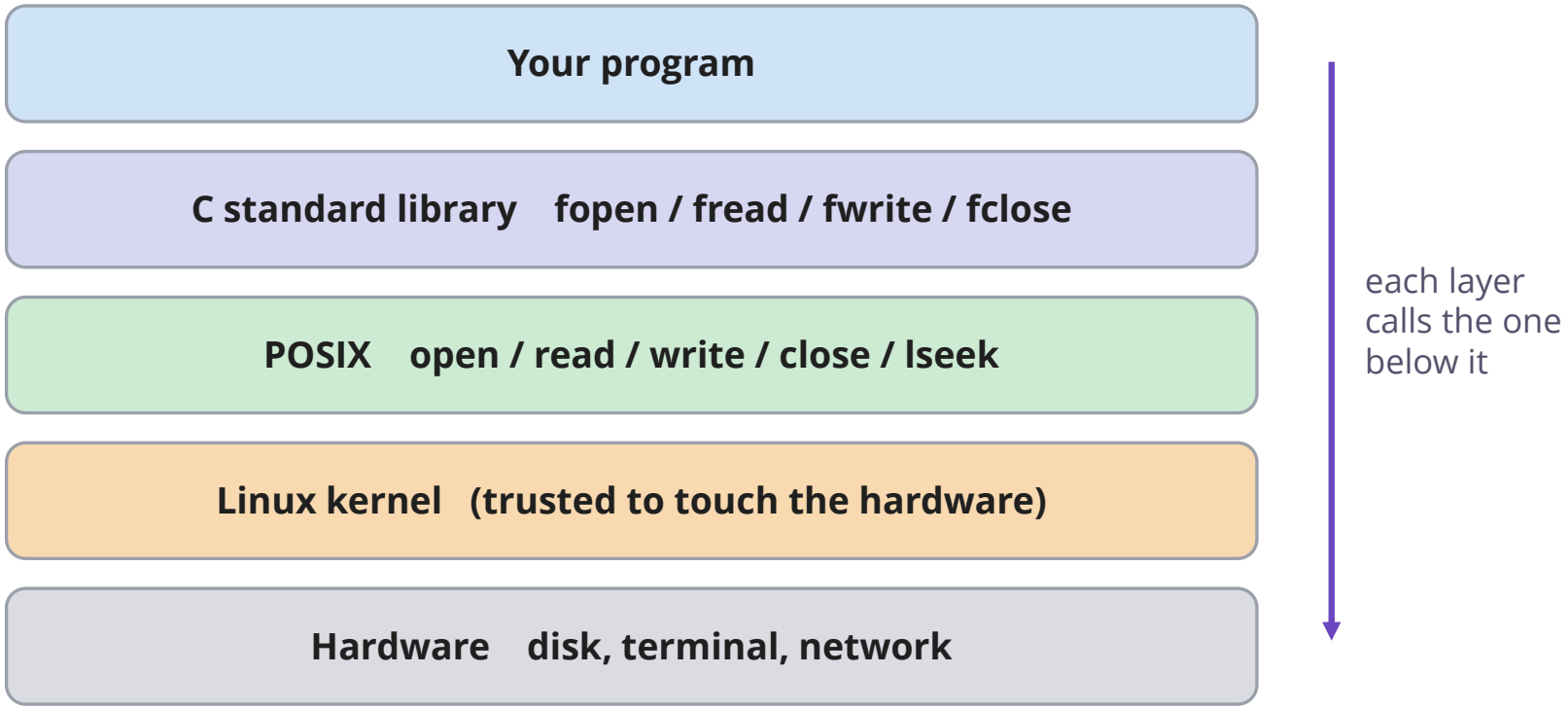
General:

- Error on questions.txt 2A/2B for Exercise 4
- Would you be interested in a guest lecture?

Review

REVIEW

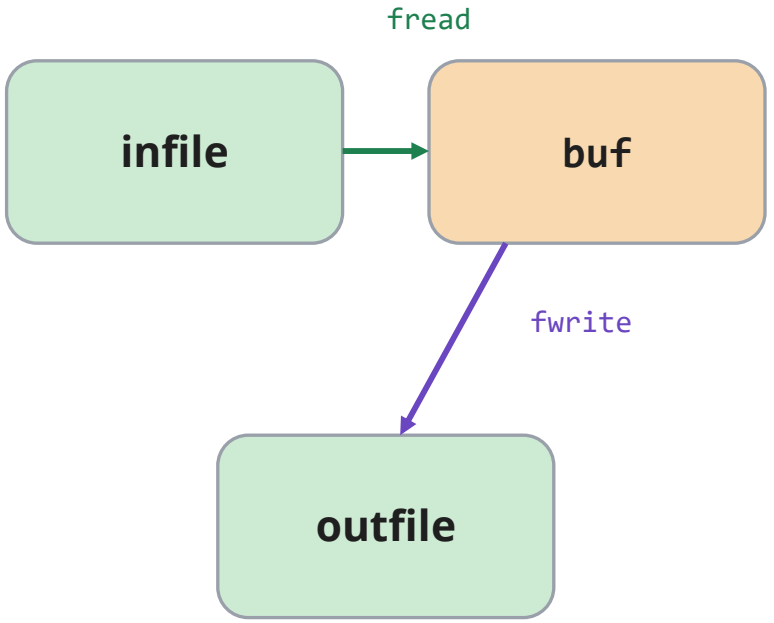
The I/O software stack



C standard library: copying a file

cp_example.c

```c
FILE* fin = fopen(argv[1], "rb");
if (fin == NULL) { return EXIT_FAILURE; }
FILE* fout = fopen(argv[2], "wb");
if (fout == NULL) {
  fclose(fin);
  return EXIT_FAILURE;
}
while ((n = fread(buf, 1, SIZE, fin)) > 0) {
  fwrite(buf, 1, n, fout);
}
fclose(fin);
fclose(fout);
return EXIT_SUCCESS;
```

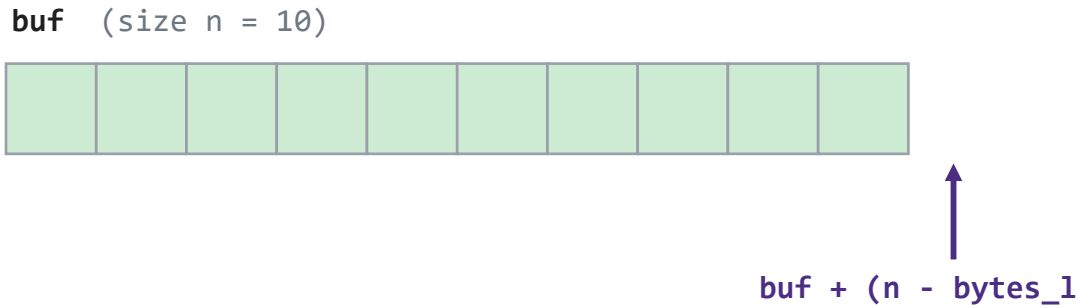


POSIX: Reading exactly n bytes

readN.c

```
1  int bytes_left = n;
2  while (bytes_left > 0) {
3    result = read(fd, buf + (n - bytes_left), bytes_left);
4    if (result == -1) {
5      if (errno != EINTR) break;   // a real error
6      continue;                    // EINTR: just retry
7    } else if (result == 0) {
8      break;                       // EOF
9    }
10   bytes_left -= result;
11 }
```

buf as the loop runs (n = 10):



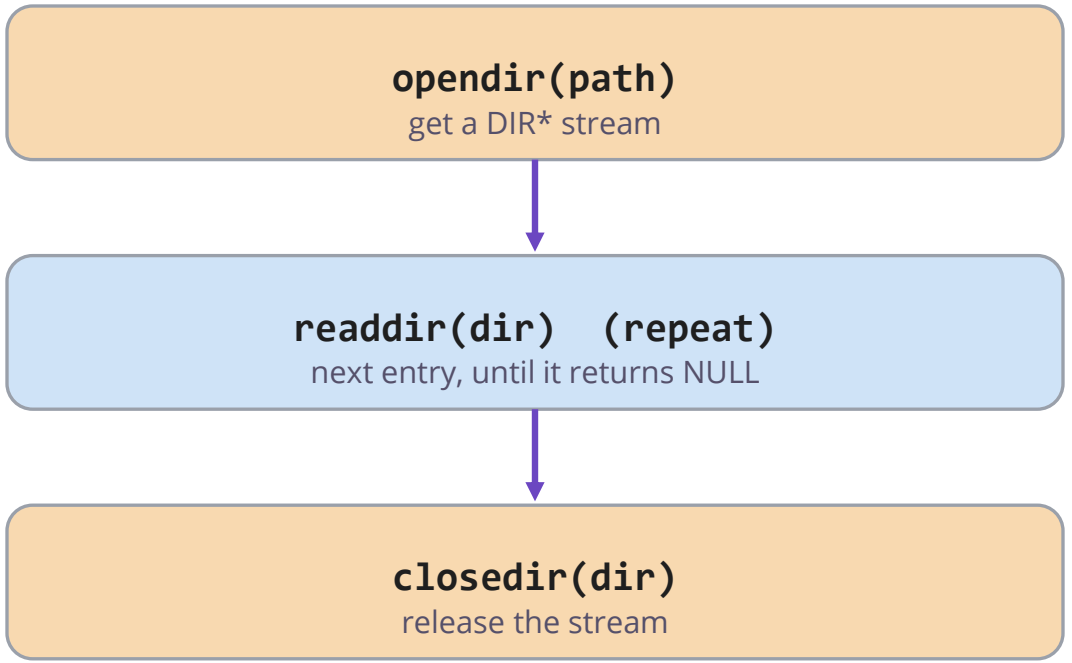
bytes_left = **0** **green = read**, **orange = todo**

Later reads fill the rest. When bytes_left hits 0 the loop ends and buf holds all n bytes, assembled from however many reads it took.

REVIEW

A directory is just a special file

- A directory is a file whose contents are a list of entries (names plus some metadata)
- So POSIX gives you directory calls that mirror open / read / close
- **You do not read the bytes yourself:** the library hands you one entry at a time



REVIEW

struct dirent

```
1  struct dirent {
2    ino_t          d_ino;     // inode number
3    off_t          d_off;     // not really an offset
4    unsigned short d_reclen;  // length of this record
5    unsigned char  d_type;    // file type (see note)
6    char           d_name[NAME_MAX + 1];  // the name
7  };
```

- **d_name** is what you almost always want
- The rest is bookkeeping the kernel uses

d_type is probably not what you think (not a file extension)

- **It is the entry's file type** (DT_REG, DT_DIR, DT_LNK, ...), not a general field. Not every filesystem fills it in; it can be DT_UNKNOWN, in which case use stat() to find the type.

POSIX: Traverse a directory

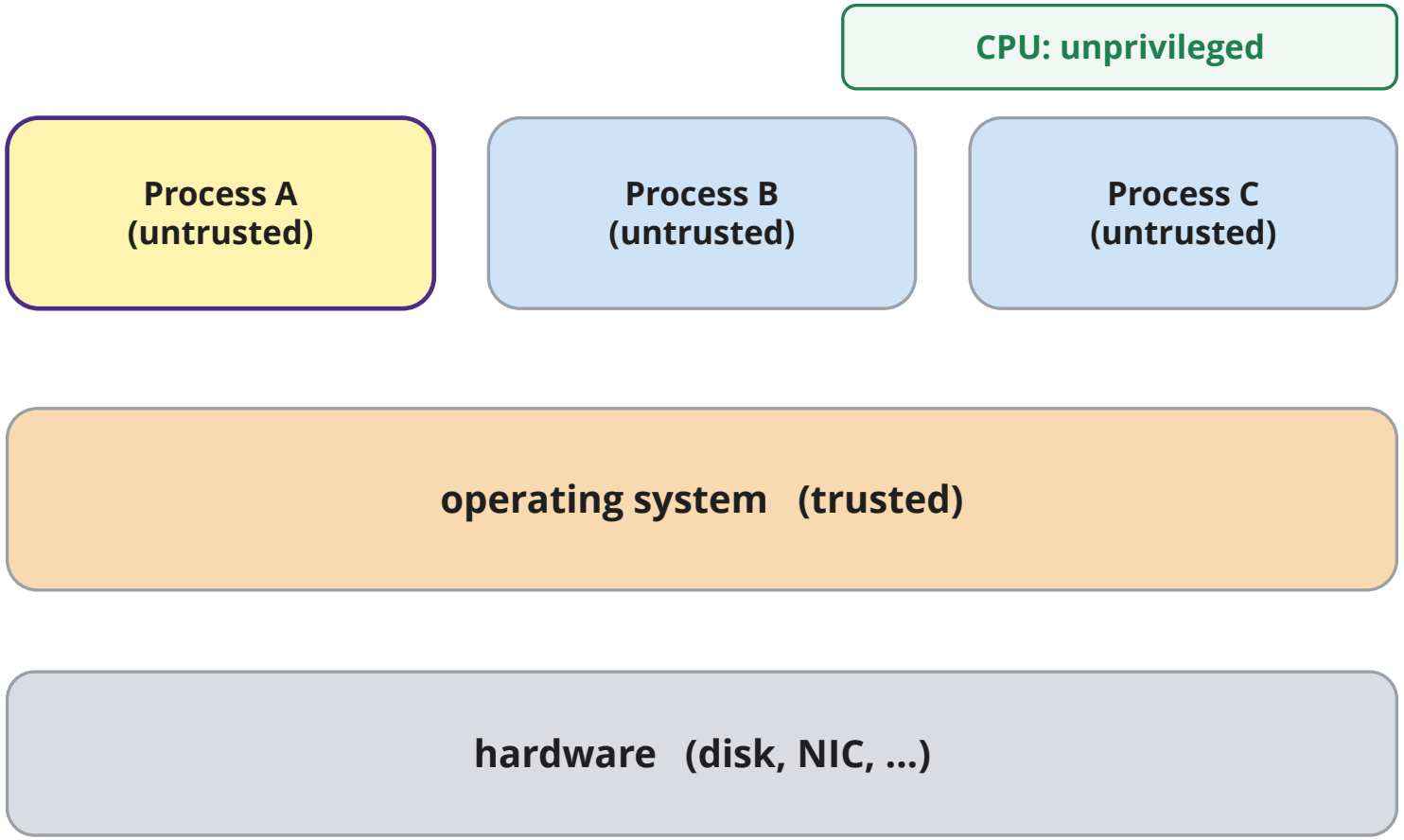
```c
#include <dirent.h>    // opendir, readdir, closedir
#include <...>

int main(int argc, char** argv) {
  DIR* dir = opendir(argv[1]);
  if (dir == NULL) {
    perror("opendir");
    return EXIT_FAILURE;
  }

  struct dirent* entry;
  while ((entry = readdir(dir)) != NULL) {
    printf("%s\n", entry->d_name);
  }
  closedir(dir);
  return EXIT_SUCCESS;
}
```

- readdir in the loop condition: stop when it returns NULL
- **Print entry->d_name** each time
- Then closedir to release the stream

```
$ ./list .
$ .
$ ..
$ list.c
$ Makefile
```

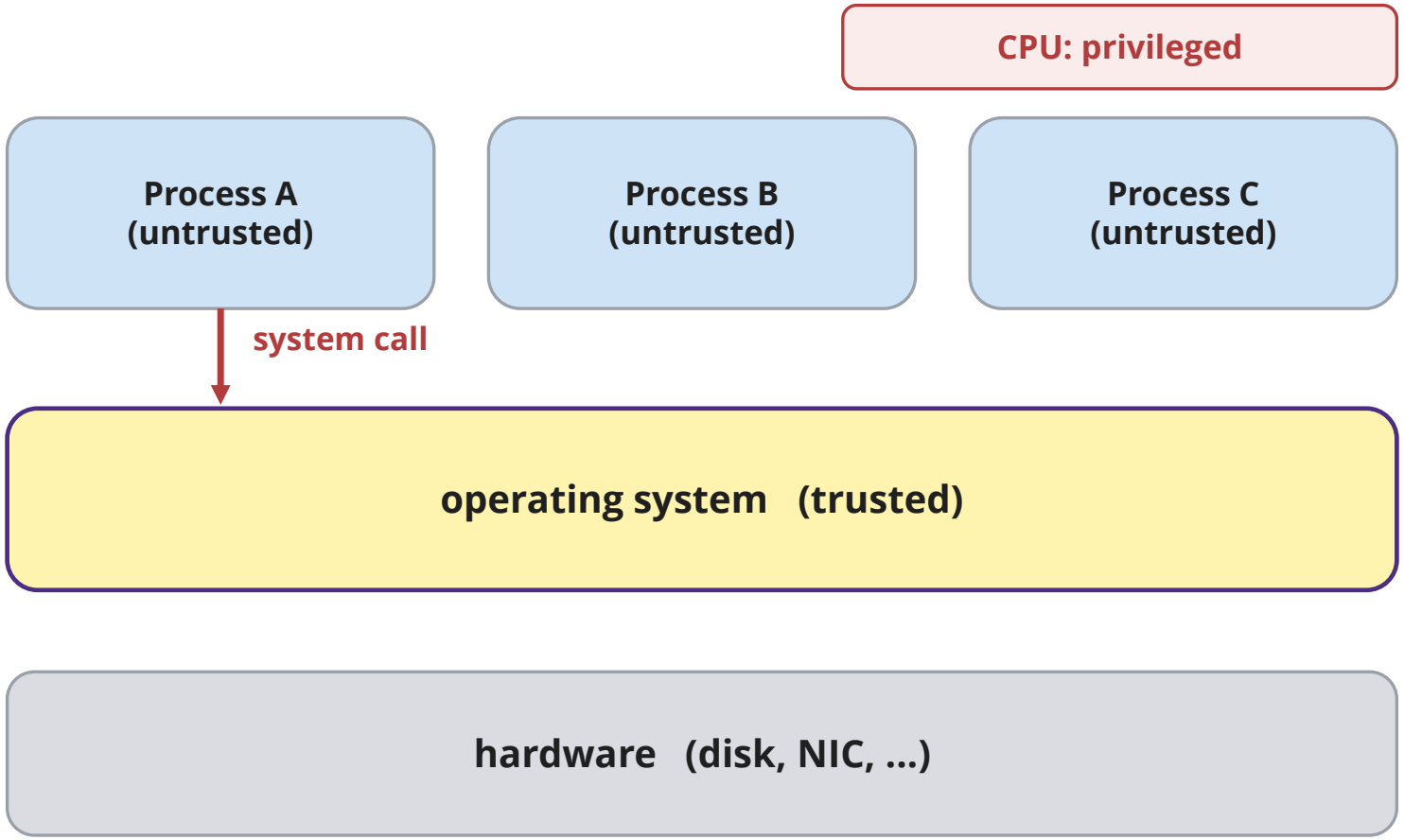
REVIEW

System calls



A CPU is running user-level code in Process A. The CPU is in unprivileged mode, so it cannot touch hardware directly.

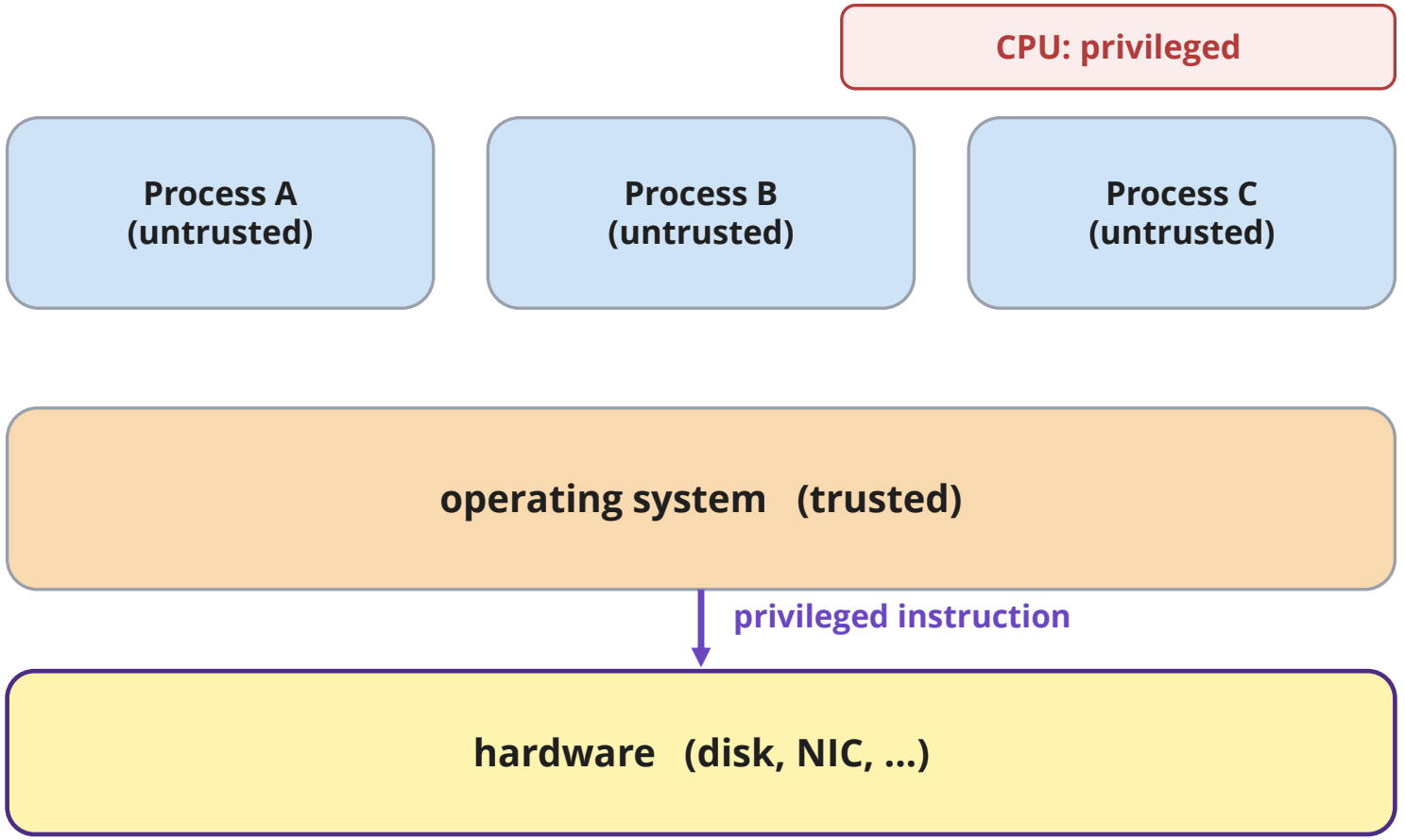
System calls



Process A invokes a system call. The hardware switches the CPU to privileged mode and traps into the OS, which runs the matching system-call handler.

REVIEW

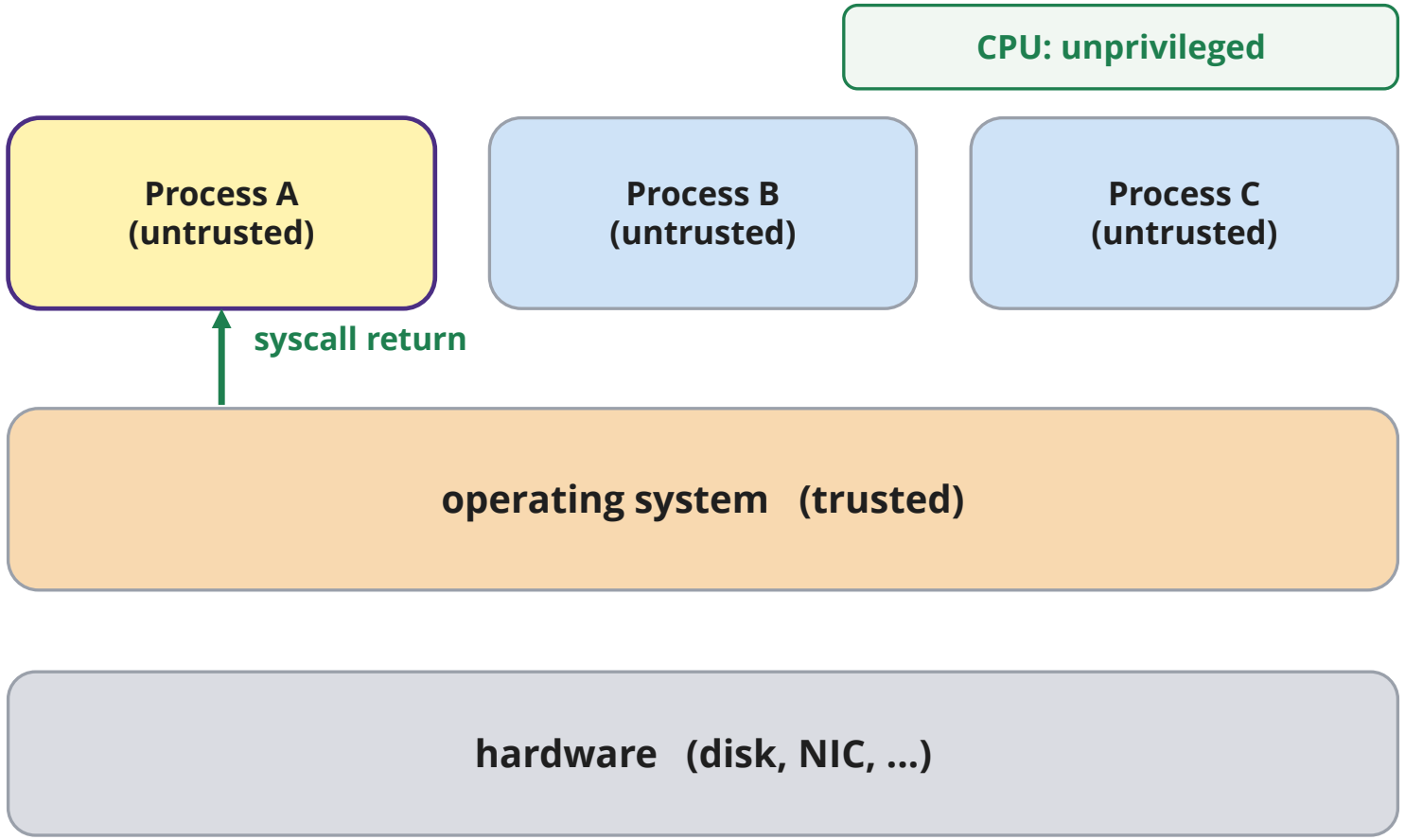
System calls



Because the CPU is now privileged, the OS can run privileged instructions that talk directly to hardware, like the disk.

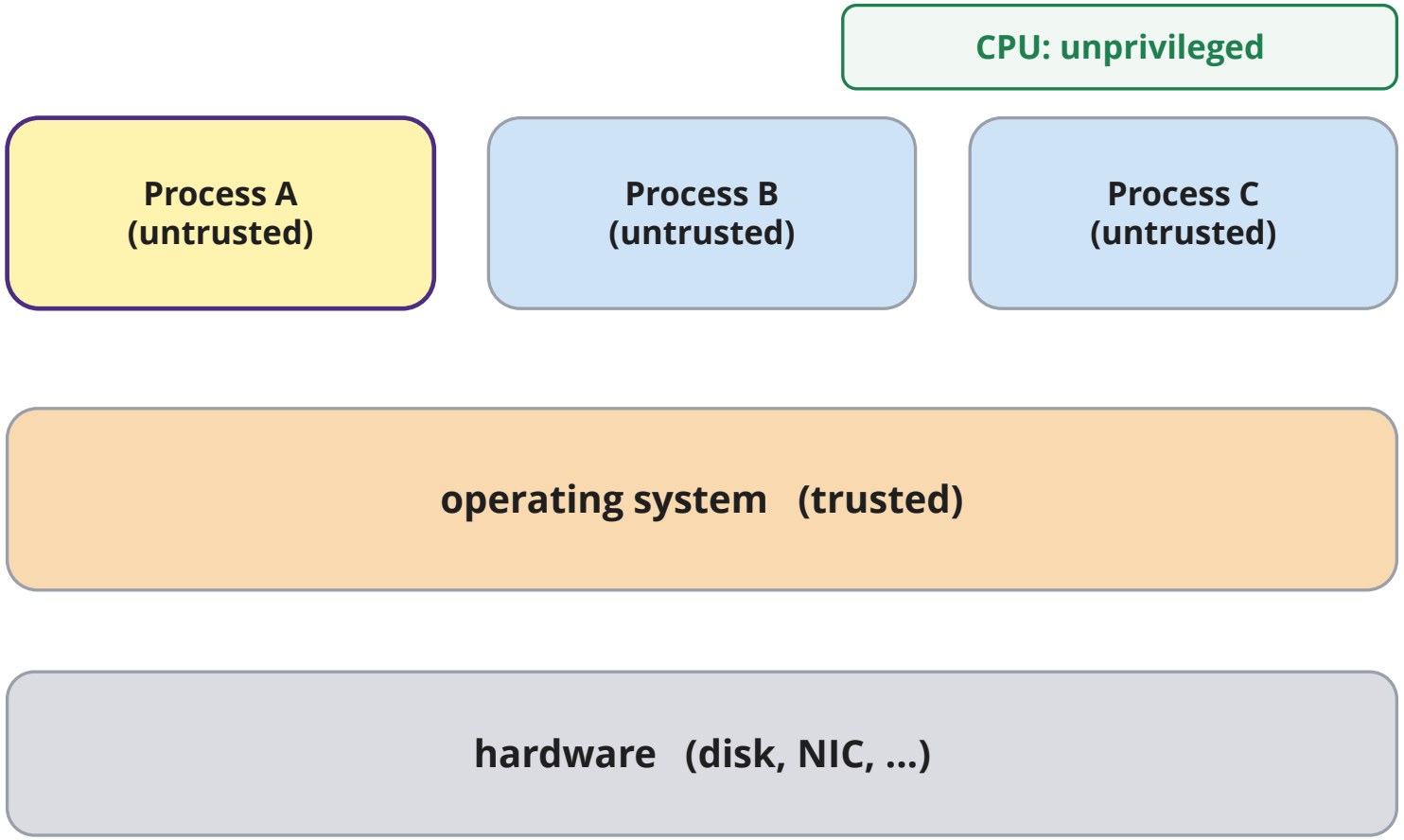
This is the slow part: hardware waits can be long, so the OS may run another process meanwhile.

System calls



When the handler finishes (possibly after a long hardware wait), the OS sets the CPU back to unprivileged mode and returns into Process A's code.

System calls



Process A carries on with whatever code came after the system call, none the wiser about the mode switches that happened.

Hello World in C

```c
#include <stdio.h>    // for printf
#include <stdlib.h>   // for EXIT_SUCCESS

int main(int argc, char** argv) {
  printf("Hello, World!\n");
  return EXIT_SUCCESS;
}
```

```
$ gcc -Wall -g -std=c17 -o hello hello.c
```

- The C you already know, compiled with gcc
- **Reminder:** that printf becomes a write system call
- Now let's write the exact same thing in C++

Hello World in C++

```cpp
#include <iostream>   // for cout, endl
#include <cstdlib>    // for EXIT_SUCCESS

int main(int argc, char** argv) {
  std::cout << "Hello, World!" << std::endl;
  return EXIT_SUCCESS;
}
```

```
$ g++ -Wall -g -std=c++17 -o hello hello.cc
```

- **Compile with g++**, and the file is **.cc**
- **Different include** (iostream), and a new-looking output line
- Same program, so let's compare the two side by side

Interoperability between C and C++

arith.h (guarded)

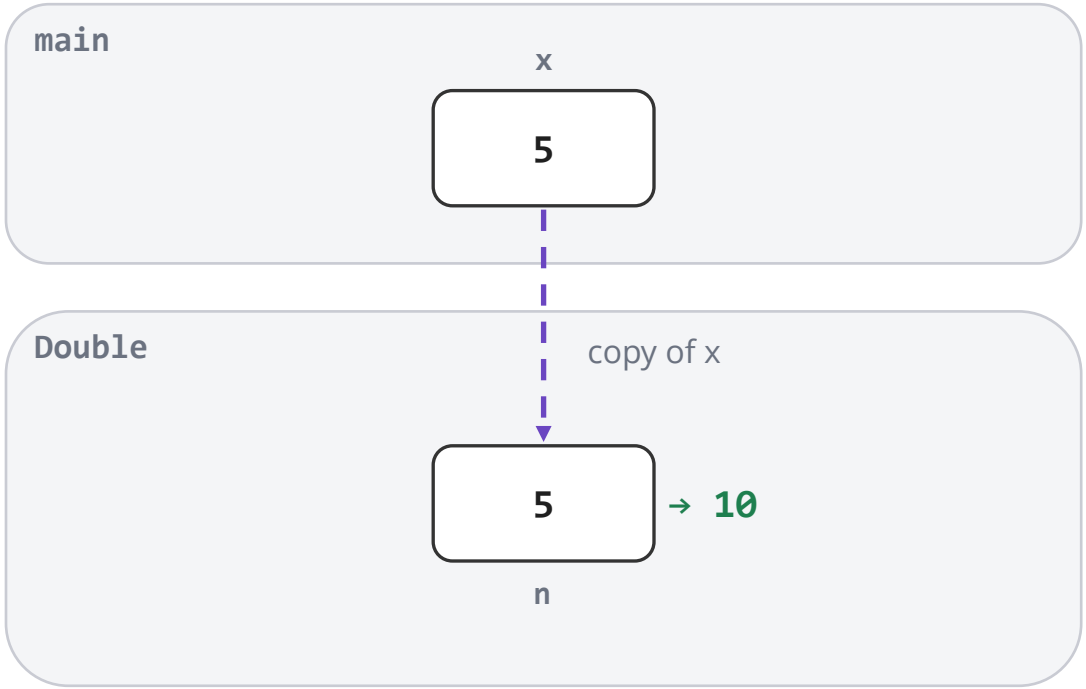
```
1  #ifndef ARITH_H_
2  #define ARITH_H_
3
4  #ifdef __cplusplus
5  extern "C" {          // C++ only
6  #endif
7
8  int Add(int a, int b);
9  int Max(int a, int b);
10
11 #ifdef __cplusplus
12 }
13 #endif
14
15 #endif  // ARITH_H_
```

- **extern "C"** tells C++: link these with plain C names
- **The __cplusplus** guard means:
  - a C++ compiler sees the extern "C" wrapper
  - a C compiler skips it (it is not valid C)
- **One header,** used unchanged by both gcc and g++

References

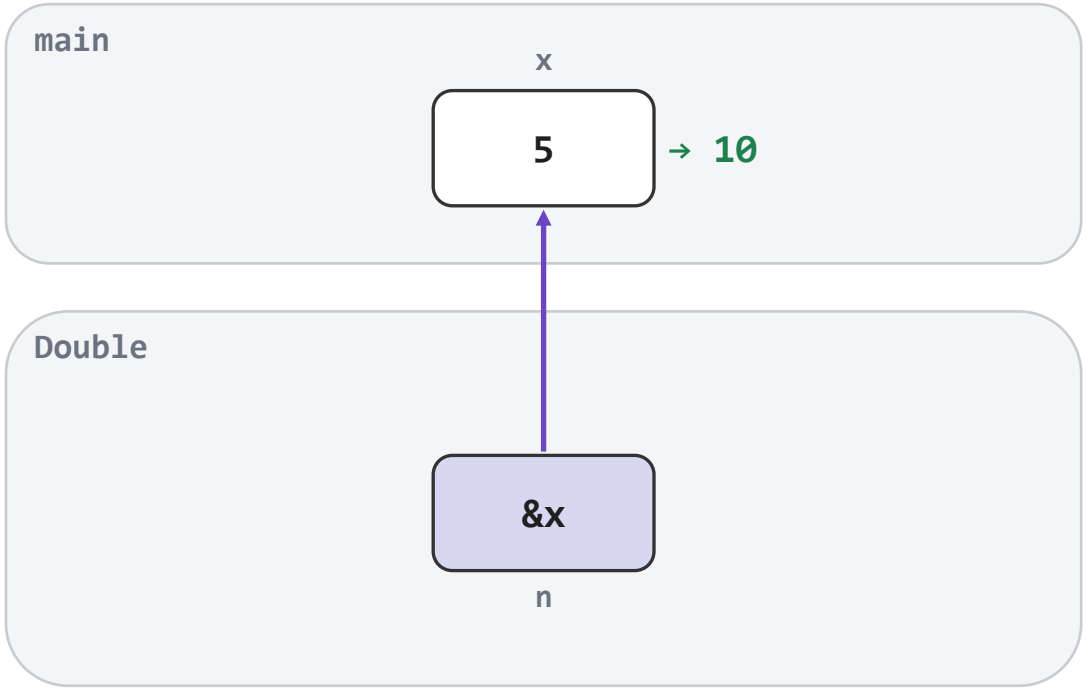
Pass-by-Value (in C)

```
1  // Tries to double n.
2  void Double(int n) {
3    n = n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    Double(x);
9    printf("x = %d\n", x);  // 5, not 10!
10   return EXIT_SUCCESS;
11 }
```



Pass-by-Reference (in C)

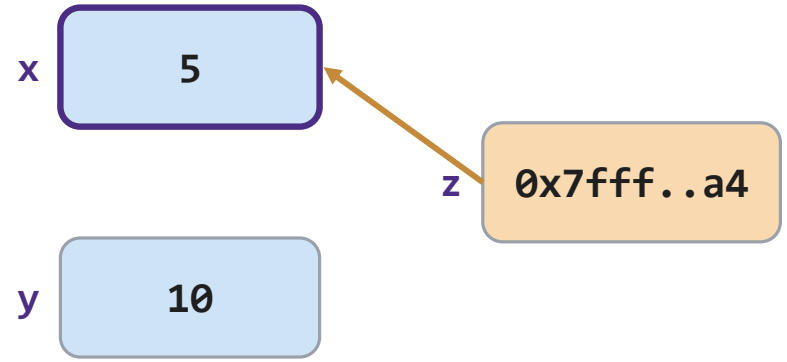
```
1  // Doubles the value n points to.
2  void Double(int* n) {
3    *n = *n * 2;
4  }
5
6  int main(int argc, char* argv[]) {
7    int x = 5;
8    Double(&x);
9    printf("x = %d\n", x);  // 10
10   return EXIT_SUCCESS;
11 }
```



Setup

Pointers (quick review)

```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int* z = &x;   // z holds the address of x
4    *z += 1;        // sets x (and *z) to 6
5     x += 1;        // sets x (and *z) to 7
6     z = &y;        // z now points at y
7    *z += 1;        // sets y (and *z) to 11
8    return EXIT_SUCCESS;
9  }
```

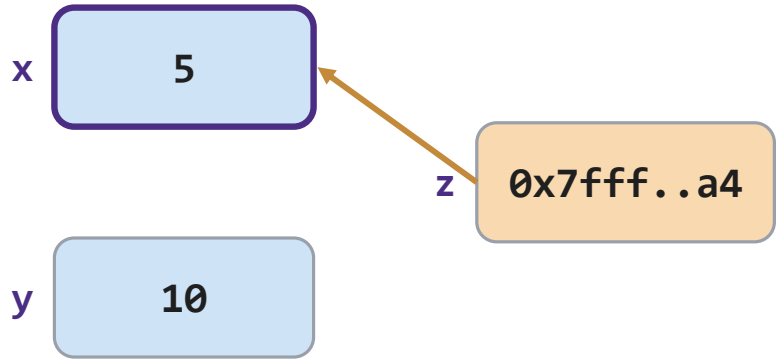


Two ints (x, y) and a pointer z aimed at x. Nothing has run yet; we will step through it one line at a time.

Step 1 of 3

Pointers (quick review)

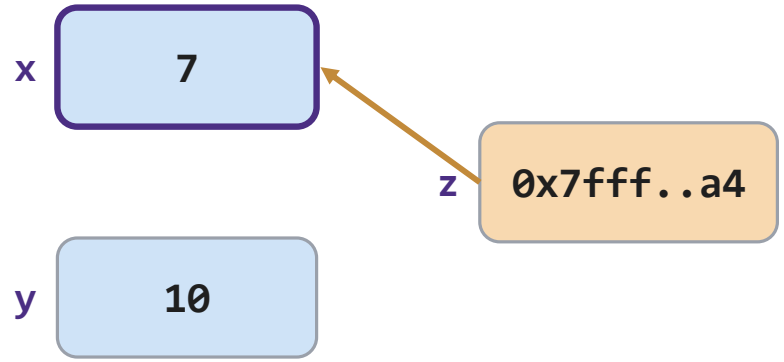
```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int* z = &x;    // z holds the address of x
4    *z += 1;         // sets x (and *z) to 6
5     x += 1;         // sets x (and *z) to 7
6     z = &y;         // z now points at y
7    *z += 1;         // sets y (and *z) to 11
8    return EXIT_SUCCESS;
9  }
```



Step 2 of 3

Pointers (quick review)

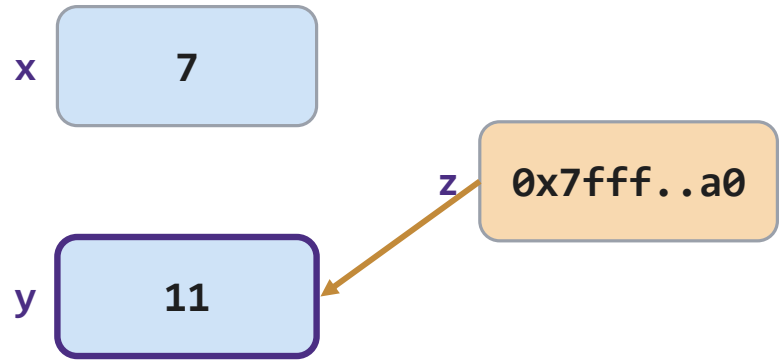
```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int* z = &x;    // z holds the address of x
4    *z += 1;         // sets x (and *z) to 6
5     x += 1;         // sets x (and *z) to 7
6     z = &y;         // z now points at y
7    *z += 1;         // sets y (and *z) to 11
8    return EXIT_SUCCESS;
9  }
```



Step 3 of 3

Pointers (quick review)

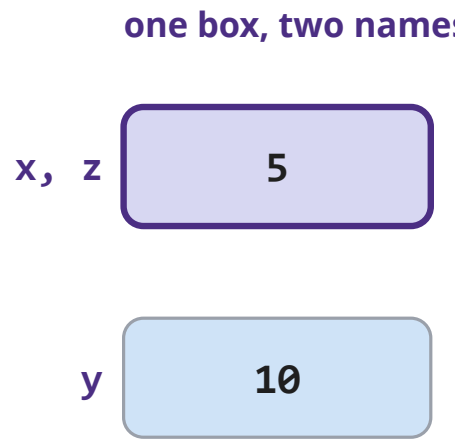
```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int* z = &x;    // z holds the address of x
4    *z += 1;         // sets x (and *z) to 6
5     x += 1;         // sets x (and *z) to 7
6     z = &y;         // z now points at y
7    *z += 1;         // sets y (and *z) to 11
8    return EXIT_SUCCESS;
9  }
```



Setup

Reference (in C++)

```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int& z = x;   // binds the name z to x
4    z += 1;        // sets z (and x) to 6
5    x += 1;        // sets x (and z) to 7
6    z  = y;        // copies y's VALUE into x
7    z += 1;        // sets z (and x) to 11
8    return EXIT_SUCCESS;
9  }
```

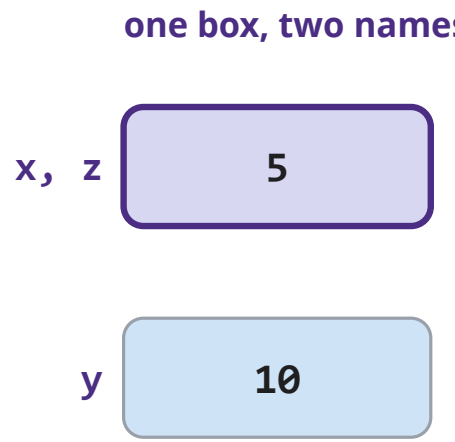


- **int& z = x;** makes z a second name for x.
- Note the single box labeled x, z. We will run it next.

Step 1 of 4

Reference (in C++)

```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int& z = x;   // binds the name z to x
4    z += 1;         // sets z (and x) to 6
5    x += 1;         // sets x (and z) to 7
6    z  = y;         // copies y's VALUE into x
7    z += 1;         // sets z (and x) to 11
8    return EXIT_SUCCESS;
9  }
```

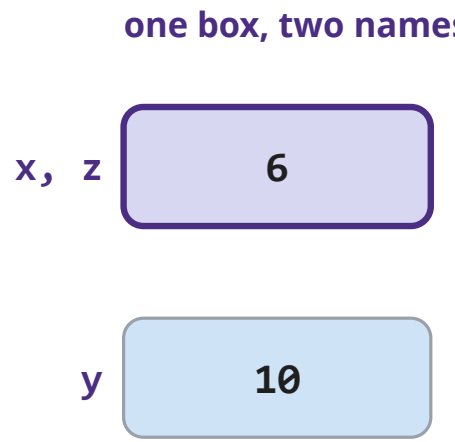


- **int& z = x;** makes z another name for x.
- There is no separate box for z, and no address to store.

Step 2 of 4

Reference (in C++)

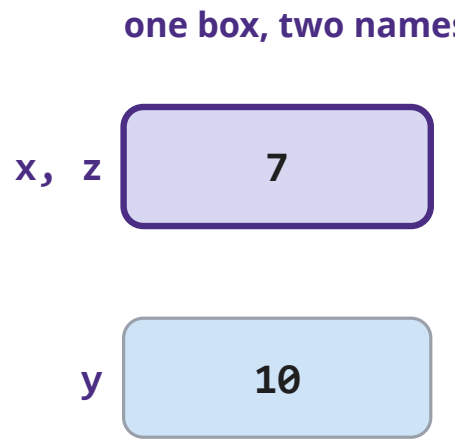
```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int& z = x;   // binds the name z to x
4    z += 1;        // sets z (and x) to 6
5    x += 1;        // sets x (and z) to 7
6    z  = y;        // copies y's VALUE into x
7    z += 1;        // sets z (and x) to 11
8    return EXIT_SUCCESS;
9  }
```



❖ **z += 1;** changes the one shared box: x and z are now 6.

Reference (in C++)

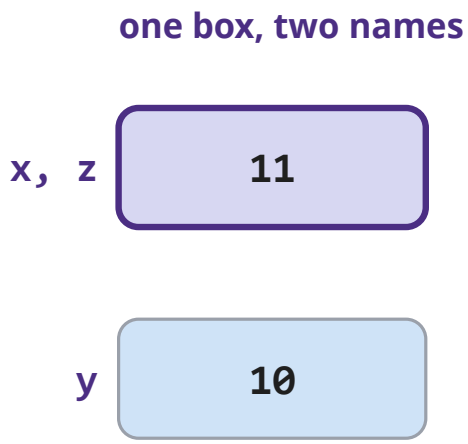
```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int& z = x;   // binds the name z to x
4    z += 1;        // sets z (and x) to 6
5    x += 1;        // sets x (and z) to 7
6    z  = y;        // copies y's VALUE into x
7    z += 1;        // sets z (and x) to 11
8    return EXIT_SUCCESS;
9  }
```



❖ **x += 1;** also changes z: both are 7. Same box, two names.

Reference (in C++)

```
1  int main(int argc, char** argv) {
2    int x = 5, y = 10;
3    int& z = x;   // binds the name z to x
4    z += 1;        // sets z (and x) to 6
5    x += 1;        // sets x (and z) to 7
6    z  = y;        // copies y's VALUE into x
7    z += 1;        // sets z (and x) to 11
8    return EXIT_SUCCESS;
9  }
```



- **z = y;** does NOT re-bind z. It copies y's value (10) into the shared box.
- **z += 1** then makes it 11; y stays 10.

Pointers vs references, side by side

```
1  void IncPtr(int* p) { *p += 1; }   // pointer version
2  void IncRef(int& r) {  r += 1; }   // reference version
3
4  int main(int argc, char** argv) {
5    int a = 5;
6    IncPtr(&a);   // caller: &a,  body: *p
7    IncRef(a);    // caller:  a,  body:  r
8    // a is now 7
9    return EXIT_SUCCESS;
10 }
```

- **Both functions do the same thing**: add 1 to the caller's a.
- **Pointer:** the caller writes **&a**, the body writes ***p**.
- **Reference:** the caller writes **a**, the body writes **r**. No stars, no ampersands.

Pass around big objects cheaply

```
1  struct Image {                  // a big struct
2    int  width, height;
3    char pixels[1000000];         // ~1 MB of data
4  };
5
6  void Save(Image img);           // copies ~1 MB!
7  void Save(Image& img);          // no copy
```

❖ **Pass by value** copies the whole struct into the parameter. Here that is a 1 MB copy on every call.

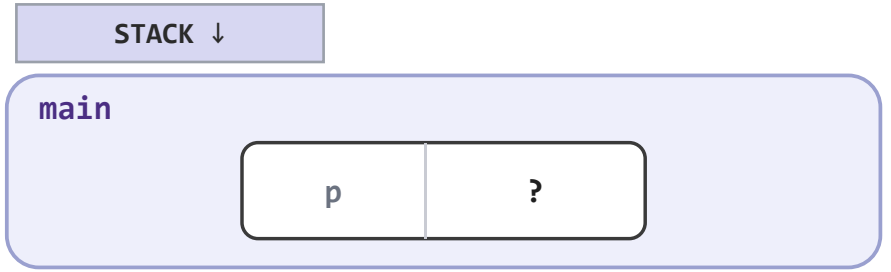
❖ Pass by **const&** hands the function a read-only alias: no copy, and the compiler forbids modifying it.

❖ **This is the most common reference idiom in C++:** big inputs go by const reference.

Takeaway: const& = pass big things without copying, while promising not to change them. It is the reason references and const almost always appear together.

Stack: dangling pointer to a local

```c
int* Bad(void) {
  int local = 42;
  return &local;   // address of a local!
}

int main(int argc, char* argv[]) {
  int* p = Bad();
  printf("%d\n", *p);  // reads dead memory
  return EXIT_SUCCESS;
}
```



main runs; about to call Bad()

Stack: dangling reference to a local

```
1  int& BrokenRef() {
2    int local = 42;
3    return local;      // BUG: local dies here
4  }
5  int main(int argc, char** argv) {
6    int& r = BrokenRef();   // aliases dead memory
7    std::cout << r << std::endl;  // undefined!
8    return EXIT_SUCCESS;
9  }
```

- A reference is not a lifetime guarantee. It is just a name for something that must outlive it.
- Returning a reference to a local **dangles** the moment the frame is popped.
- Same hazard as returning a pointer to a local in C.
- Rule: never return a reference (or pointer) to a local.

const

In practice… we normally use this instead

```
1  struct Image {                  // a big struct
2    int  width, height;
3    char pixels[1000000];         // ~1 MB of data
4  };
5
6  void Save(Image img);           // copies ~1 MB!
7  void Save(const Image& img);    // no copy
```

const: a promise this will not change

```
1  const int kMaxHealth = 100;   // a promise: fixed
2
3  int hp = kMaxHealth;   // ok: reading is fine
4  kMaxHealth = 120;      // error: cannot modify
```

- **const** means "I promise not to modify this."
- Reading a const value is fine; writing to it is a compile error.
- We just used it as **const Image&**: pass without copying, promise not to change.
- Used far more in C++ than in C.

Takeaway: const asks the compiler to block assignment!

const pointer

```
1  Account acc   = {7, 100};
2  Account other = {9, 500};
3
4  Account* const p = &acc;  // a const POINTER
5  p->balance = 0;   // ok: change the data
6  p = &other;       // error: pointer is fixed
```

- A pointer touches two separate things:
  - **the data** it points at (**p->balance**)
  - **its own value** (which object, **p = …**)
- **Account* const p** locks the pointer itself.
  - **p->balance = 0** is fine (data is free)
  - **p = &other** is an error (pointer is fixed)

const type*

```
1  Account acc   = {7, 100};
2  Account other = {9, 500};
3
4  const Account* p = &acc;  // const moved to front
5  p->balance = 0;   // ??  does this compile?
6  p = &other;       // ??  does this compile?
```

- **We moved const to the front.**
- Same two questions as before:
  - can we do **p->balance = 0**?
  - can we do **p = &other**?
- Talk to your neighbor. Which one flips, and why?

Predict first: decide what each line does before we reveal the rule on the next slide.

const type*

```
1  const Account acc   = {7, 100};
2  Account other = {9, 500};
3
4  const Account* p = &acc;  // const moved to front
5  p->balance = 0;   // ??  does this compile?
6  p = &other;       // ??  does this compile?
```

- **We moved const to the front.**
- Same two questions as before:
  - can we do **p->balance = 0**?
  - can we do **p = &other**?
- Talk to your neighbor. Which one flips, and why?

Read the declaration right to left

```
const Account* p;
```

read: p is a pointer to a const Account

data locked, pointer free

`p->balance=0;` error `p=&other;` ok

```
Account* const p;
```

read: p is a const pointer to an Account

pointer locked, data free

`p->balance=0;` ok `p=&other;` error

```
const Account* const p;
```

read: const pointer to a const Account

both locked

`p->balance=0;` error `p=&other;` error

Make parameters const when you can

```
1  void Print(const Account* a) {  // promises: no change
2    std::cout << a->balance << std::endl;
3  }
4  void Reset(Account* a) {        // may modify *a
5    a->balance = 0;
6  }
7  int main(int argc, char** argv) {
8    const Account locked = {7, 100};
9    Account open = {9, 500};
10   Print(&locked);   // OK
11   Print(&open);     // OK
12   Reset(&locked);   // error: breaks const
13   Reset(&open);     // OK
14   return EXIT_SUCCESS;
15 }
```

❖ **Print** promises not to change *a, so it accepts both a const and a non-const Account.

❖ **Reset** might change *a, so the compiler refuses a const argument (that would break the const promise).

❖ **Style tip:** mark parameters const whenever the function does not need to modify them. It documents intent and widens what callers can pass.

When to use references (Google style)

- **Input parameters:**
  - values for small things (int, small structs)
  - const references for big objects (no copy, read-only)
- **Output parameters:**
  - const pointers: unchangeable pointer to changeable data
  - passing **&x** at the call site flags "this may be written"
- **Ordering:** inputs first, outputs last.

```
1  // input by const ref, output by const pointer
2  void Transfer(const Account& from,   // input
3                Account* const into) { // output
4    into->balance += from.balance;
5  }
```

Takeaway: Inputs are values or const references; outputs are const pointers; list inputs before outputs. A caller's & then reads as "this argument may change."

CHECK YOUR UNDERSTANDING

What does this print?

```
1  void Foo(int& x, int* y, int z) {
2    z = *y;
3    x += 2;
4    y = &x;
5  }
6  int main(int argc, char** argv) {
7    int a = 1;
8    int b = 2;
9    int& c = a;
10   Foo(a, &b, c);   // ?
11   std::cout << "(" << a << ", " << b
12             << ", " << c << ")";
13   return EXIT_SUCCESS;
14 }
```

- **x** is a reference parameter.
- **y** is a pointer, passed by value.
- **z** is a plain int, passed by value.
- **c** is a reference to a.
- A. (1, 2, 3)
- B. (3, 2, 3)
- C. compiler error

Predict first: trace each parameter's binding before you vote, then we reveal it on the next slide.

BONUS

const with auto

```
1  int x = 5;
2  const int y = 6;
3
4  auto a = y;          // a is int  (const dropped!)
5  const auto b = y;    // b is const int
6  const auto& c = x;   // c is a const int& to x
```

❖ **auto** deduces the value type and drops top-level const and references by default.

❖ **auto a = y** gives a plain int copy, not a const int.

❖ Add const and & back yourself when you want them: const auto& is the common read-only, no-copy form.

Takeaway: auto copies by value and strips top-level const and references. Write const auto& when you want a cheap, read-only view.

Reminders

Assessments:

- Exercise 4 will be due Friday at 11:59pm
  - Includes C and C++
- HW2 will be released tomorrow
  - Due next Thursday at 11:59pm

General:

- Error on questions.txt 2A/2B for Exercise 4
- Would you be interested in a guest lecture?

Questions?

Thanks for coming - see you next lecture!