CS 112

Introduction to Data Structures

WEEK 01

Control Structures, Functions, Parameter Passing

Eric Araújo

Calvin University · Fall 2026

This Week

1. Selection: `if` and `switch`
2. Loops: `while`, `for`, `break` / `continue`
3. Defining and calling functions
4. Prototypes, default arguments, overloading
5. Arrays as parameters
6. Parameter passing: in, out, in/out **KEY**
7. Building with `make`

Selection: `if`

```
if (hp <= 0) {
    cout << "You have fallen." << endl;
} else if (hp < 20) {
    cout << "You are badly hurt." << endl;
} else {
    cout << "You fight on." << endl;
}
```

- Braces are optional for one statement: **use them anyway**
- Boolean operators: `==` `!=` `&&` `||` `!`
- `=` is assignment, `==` is comparison: the classic C++ bug

> Format as you go: in VS Code, **F1 → Format Document**, or `Shift`+`Alt`+`F` (`Shift`+`Option`+`F` on a Mac).
> Readable code is hospitality toward whoever reads it next.

Selection: `switch`

```
switch (roll) {
  case 1:
    cout << "Critical miss!" << endl;
    break;
  case 19:
  case 20:                    // 19 and 20 share a case
    cout << "Critical hit!" << endl;
    break;
  default:
    cout << "A normal swing." << endl;
}
```

> Forget `break` and execution *falls through* into the next case. Sometimes useful, usually a bug.

TALK TO YOUR NEIGHBOR · TTYN

What prints when `roll` is 19?

```
case 19:
case 20:
    cout << "Critical hit!" << endl;
    break;
default:
    cout << "A normal swing." << endl;
```

A. Critical hit!

B. Critical hit! then A normal swing.

C. A normal swing.

D. Nothing: case 19 is empty.

TALK TO YOUR NEIGHBOR · TTYN

What is printed?

```
int tier = 2;
int gold = 0;

switch (tier) {
  case 3:  gold += 100;   // no break
  case 2:  gold += 50;    // no break
  case 1:  gold += 10;
           break;
  default: gold = -1;
}
cout << gold;
```

A. 50

B. 60

C. 160

D. 10

E. -1

TALK TO YOUR NEIGHBOR · TTYN

What is printed?

```
char grade = 'X';

switch (grade) {
  case 'A': cout << "4.0 "; break;
  default:  cout << "0.0 ";
  case 'B': cout << "3.0 "; break;
  case 'C': cout << "2.0 "; break;
}
```

A. 0.0

B. 0.0 3.0

C. 0.0 3.0 2.0

D. Nothing: `default` must come last

E. It does not compile

TALK TO YOUR NEIGHBOR · TTYN

Does this compile?

```
string charClass = "mage";

switch (charClass) {
  case "mage":   cout << "Fireball!"; break;
  case "cleric": cout << "Heal!";     break;
}
```

A. Yes: it prints Fireball!

B. Yes, but it prints nothing

C. No: `switch` needs a whole-number type

D. No: every `case` needs a `break`

Loops: `while`

```
// how many times can we halve this before reaching 1?
double value = 45444443442345345.0;
int    count = 0;

while (value > 1.0) {
    value /= 2.0;
    ++count;
}
cout << "2^" << count << " clears it" << endl;
```

> A `while` loop tests *before* each pass. If the condition starts false, the body never runs.

TALK TO YOUR NEIGHBOR · TTYN

What does `arr` hold after this runs?

```
int arr[] = { 3, 7, 11, 13 };
int i = 1;
while (i < 4) {
    arr[i] = arr[i] * 2;
    ++i;
}
```

A. 6, 14, 22, 26

B. 3, 14, 22, 26

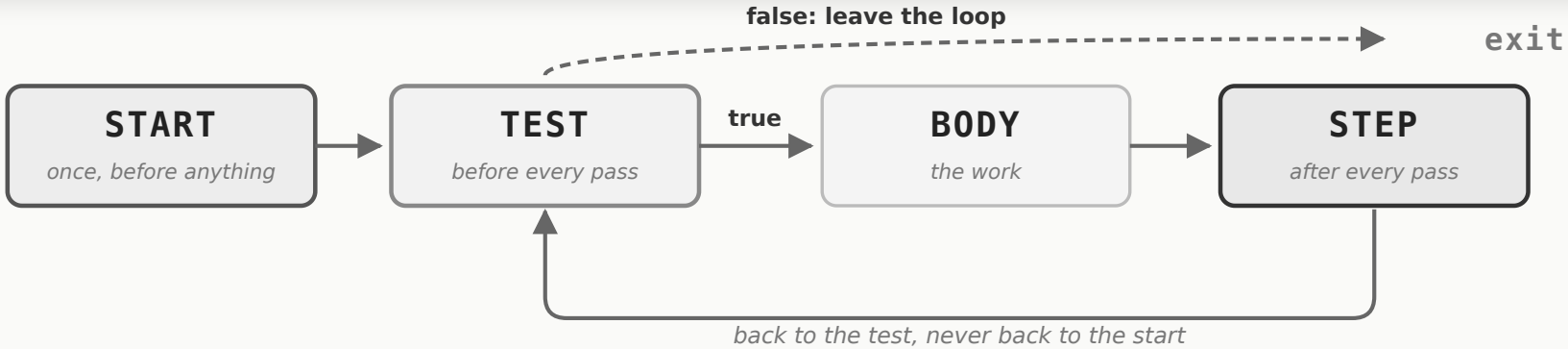
C. 6, 14, 22, 13

D. 3, 14, 22, 13

E. None of the others

From `while` to `for`

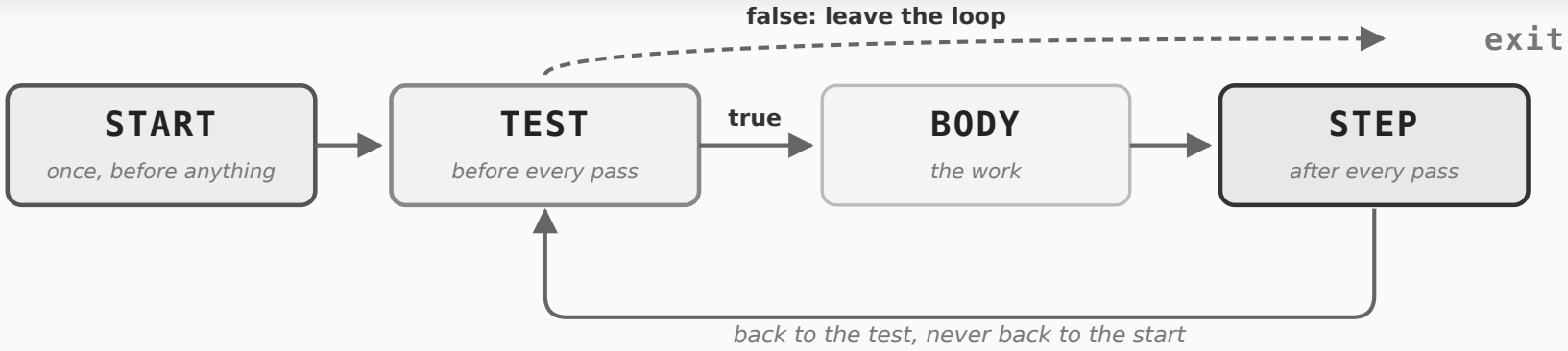
```
int i = 1;
while (i < 4) {
    arr[i] = arr[i] * 2;
    ++i;
}
```



> Three jobs, scattered over four lines. Forget the **step** and the loop never ends.

From `while` to `for`

```
for (int i = 1; i < 4; ++i) {
    arr[i] = arr[i] * 2;
}
```



Same three jobs, same order, now gathered into one header: `for (start; test; step)`.

`i` lives only inside the loop, so using it afterwards is a compile error.

TALK TO YOUR NEIGHBOR · TTYN

What is printed?

```
int total = 0;

for (int i = 1; i <= 3; ++i);
{
    total += 10;
}

cout << total;
```

A. 30

B. 10

C. 0

D. It loops forever

E. It does not compile

`break` and `continue`

break: leave the loop

```
for (int i = 0; i < n; ++i) {
  if (party[i] == target) {
    found = i;
    break;       // stop looking
  }
}
```

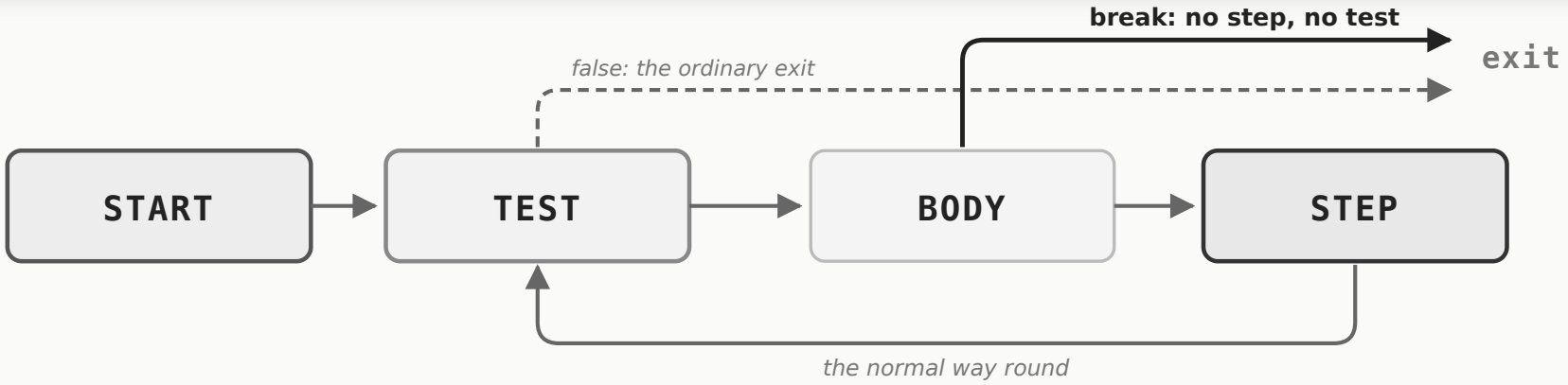
continue: skip this pass

```
for (int i = 0; i < n; ++i) {
  if (hp[i] <= 0) {
    continue;    // skip the fallen
  }
  total += hp[i];
}
```

> Both are shortcuts, not magic: anything they do can be written with a richer condition. Use them when they make the intent *clearer*.

`break`: leave the loop

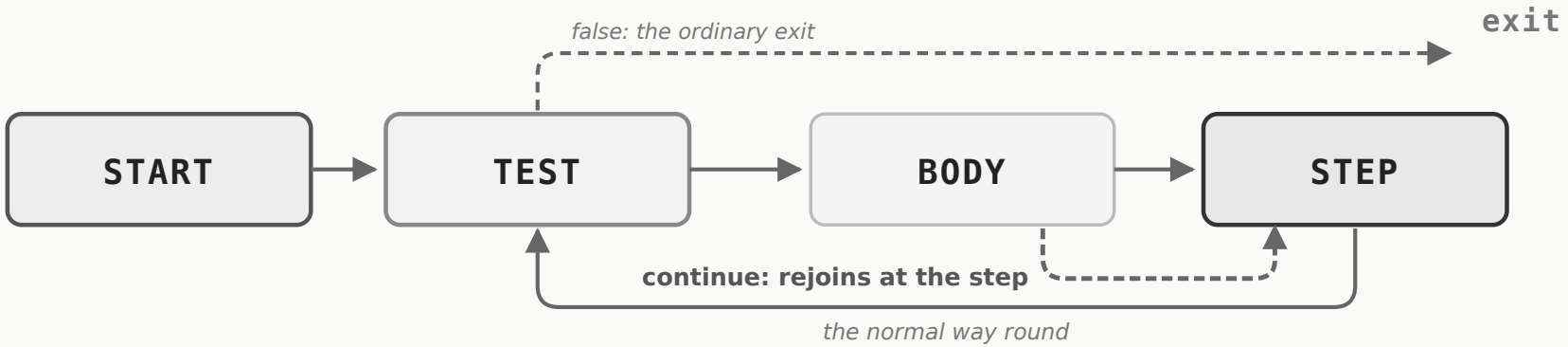
```
for (int i = 0; i < n; ++i) {
    if (party[i] == target) {
        found = i;
        break;      // stop looking
    }
}
```



The moment the target is found there is nothing left to look for, so **`break`** abandons the loop entirely: the step never runs and the test is never asked again.

`continue`: skip this pass

```
for (int i = 0; i < n; ++i) {
    if (hp[i] <= 0) {
        continue;   // skip the fallen
    }
    total += hp[i];
}
```



`continue` abandons only the rest of the body, then rejoins at the **step**. Careful: a `while` has no step, so `continue` skips a counter at the bottom of the body and the loop hangs.

Defining a Function

```
 1  int rollDamage(int dice, int sides) {
 2      int total = 0;
 3      for (int i = 0; i < dice; ++i) {
 4          total += rand() % sides + 1;
 5      }
 6      return total;
 7  }
 8
 9  int main() {
10      int hit = rollDamage(2, 6);      // 2d6
11  }
```

Parameters line 1

`int dice, int sides` — a name and a type each, comma separated. They are local variables of the function.

Arguments line 10

`2, 6` — the values you pass at the call. Copied into the parameters, in order.

The Compiler Reads Top to Bottom

```
 1  int main() {
 2      int hit = rollDamage(2, 6);
 3      cout << hit << endl;
 4  }
 5
 6  int rollDamage(int dice, int sides) {
 7      int total = 0;
 8      for (int i = 0; i < dice; ++i) total += rand() % sides + 1;
 9      return total;
10  }
```

So what happens when I compile this?

A Prototype Is a Promise

Put the signature at the top and the compiler knows what to expect long before it meets the body.

```
int rollDamage(int dice, int sides);
```

what it hands back — `int`

the name callers use — `rollDamage`

what it expects, in order — `(int dice, int sides)`

no body: that comes later — `;`

↓ further down, or in another file, the definition repeats it exactly ↓

```
int rollDamage(int dice, int sides) { /* the body */ }
```

> Return type, name and parameter **types** must match. Parameter *names* need not: `int rollDamage(int, int);` is a legal prototype, though naming them documents the order.

Where the Two Halves Live

dice.h

```
int rollDamage(int dice,
               int sides);
```

The promise. Anyone who includes this may call the function.

dice.cpp

```
#include "dice.h"

int rollDamage(int dice,
               int sides) {
    int total = 0;
    for (int i = 0; i < dice; ++i)
        total += rand() % sides + 1;
    return total;
}
```

The body. Compiled once, on its own.

main.cpp

```
#include "dice.h"

int main() {
    int hit = rollDamage(2, 6);
}
```

Sees only the promise, never the body, and that is enough to compile.

> Same idea as week 0: the header says *what exists*, the source says *how it works*, and the linker joins them at the end.

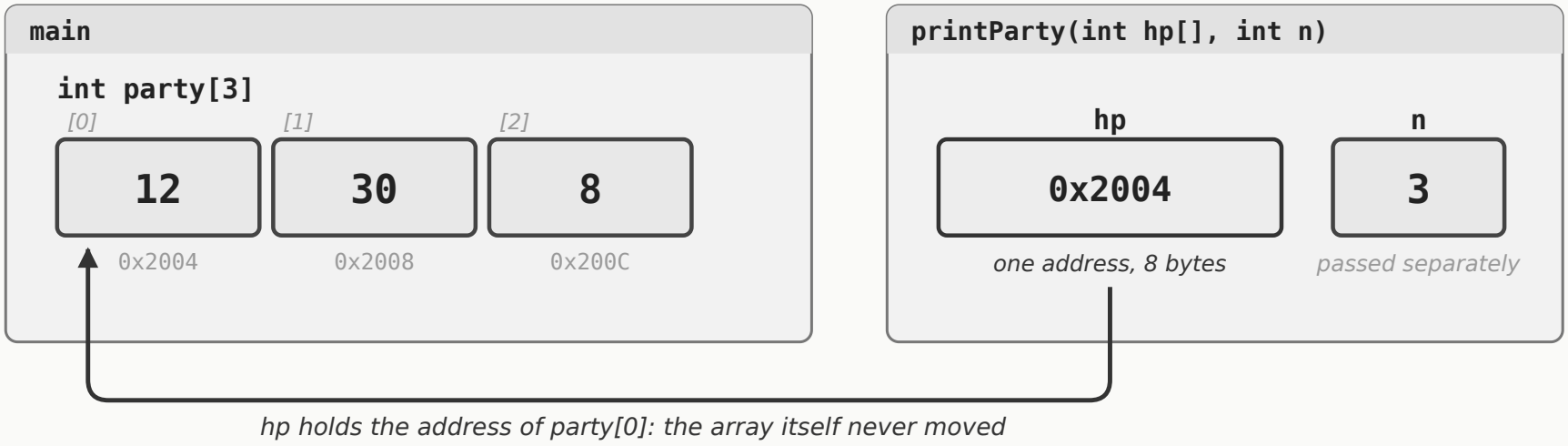
Default Arguments & Overloading

```
// default: callers may omit sides
int rollDamage(int dice, int sides = 6);

rollDamage(2);         // 2d6
rollDamage(2, 20);     // 2d20

// overloading: same name, different parameter lists
int  attack(int strength);
int  attack(int strength, int bonus);
void attack(const string &spell);
```

- Defaults go on the **prototype**, not the definition
- Overloads must differ in **parameters**: return type alone is not enough
- The compiler picks the match; ambiguity is a compile error

Passing Arrays

```
void printParty(int hp[], int n);   // int *hp means the same thing

int party[] = { 12, 30, 8 };
printParty(party, 3);               // nothing is copied
```



An array argument **decays to a pointer** to its first element, so the function cannot ask how long it is. That is why `n` has to travel with it.

TALK TO YOUR NEIGHBOR · TTYN

Which call passes `vals` more efficiently?

```
void print(int *arr, int size);
void print2(int arr[], int size);

int vals[] = { 1, 3, 4 };
print(vals, 3);
print2(vals, 3);
```

A. `print()` is more efficient

B. `print2()` is more efficient

C. Neither: they are the same

TALK TO YOUR NEIGHBOR · TTYN

Will the second call to `print2()` compile?

```
void print2(int arr[], int size);

int vals[] = { 1, 3, 4 };
int val = 7;
print2(vals, 3);
print2(&val, 1);
```

A. Yes: `&val` is an address, and `int arr[]` takes an address

B. No: `print2()` requires an actual array

C. No: you cannot have an array of size 1

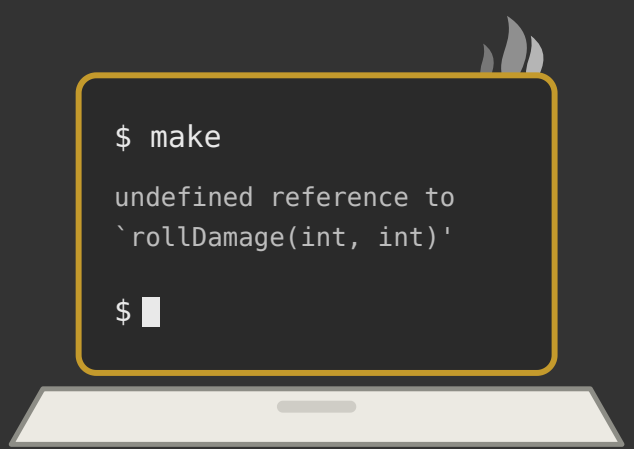
D. None of the above

Protecting an Array: `const`

```
int totalHP(const int hp[], int size) {
    int total = 0;
    for (int i = 0; i < size; ++i) {
        total += hp[i];
    }
    // hp[0] = 99;   ❌ compile error: hp is const here
    return total;
}
```

> Marking a parameter `const` is a promise to the caller: *this function will not modify your data.* The compiler enforces the promise.

Demo

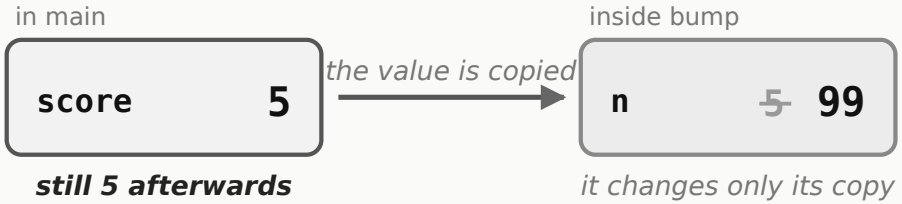


Writing a function the linker can actually find.

(declared with total confidence, defined never)

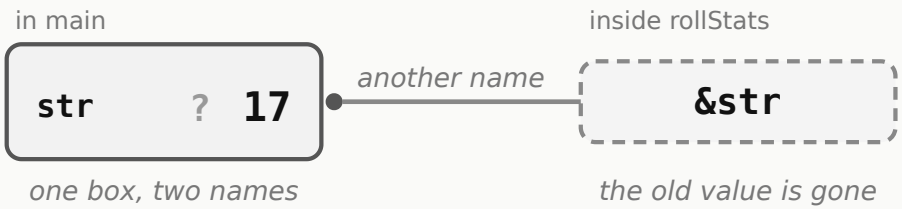
Which Way Does the Data Flow?

IN



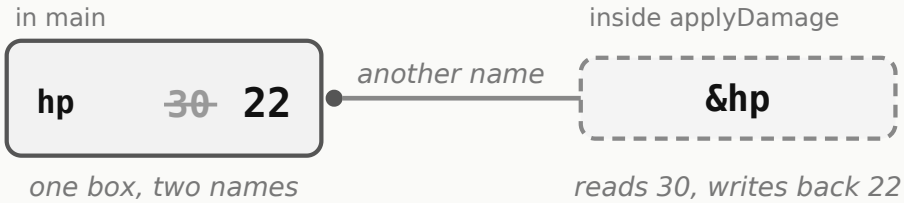

```
// IN: the function gets a copy
void bump(int n) { n = 99; }
int score = 5; bump(score);  // score: 5
```

OUT




```
// OUT: the function fills it in
void rollStats(int &str, int &dex);
int str, dex; rollStats(str, dex);
```

IN/OUT




```
// IN/OUT: reads it, writes it back
void applyDamage(int &hp, int hit);
int hp = 30; applyDamage(hp, 8);  // 22
```

A copy is a **second box**, so the caller never sees the change. A reference is a **second name for the same box**, so the caller sees everything. Pick the direction first and the mechanism follows.

Three Ways to Pass

Mechanism	Syntax	Copies?	Can change caller's value?
By value	`int hp`	Yes	No
By reference	`int &hp`	No	**Yes**
By const reference	`const string &name`	No	No

Small and IN? Pass by value, copying an `int` is free.

Big and IN? Pass by const reference, no copy, still safe.

> Anything OUT or IN/OUT must be by **plain reference**: that's the only way the caller sees the change.

TALK TO YOUR NEIGHBOR · TTYN

Does x flow *in*, *out*, or *in/out* of `j()`?

```
int j(int val) {
    val = val + 1;
    return val;
}

int main() {
    int x = 3;
    int y = j(x);
}
```

A. In

B. In/Out

C. Out

D. Out, because of what j() returns

TALK TO YOUR NEIGHBOR · TTYN

Does x flow *in*, *out*, or *in/out* of `j()`?

```
void j(int &val) {
    val = val + 1;
}

int main() {
    int x = 3;
    j(x);
}
```

A. In

B. In/Out

C. Out

D. Out, because of what j() returns

TALK TO YOUR NEIGHBOR · TTYN

Does x flow *in*, *out*, or *in/out* of `j()`?

```
void j(int &val) {
    val = 17;
}

int main() {
    int x = 3;
    j(x);
}
```

A. In

B. In/Out

C. Out

D. Out, because of what j() returns

EXTRA · READ ON YOUR OWN

Building with `make`

The next two slides are yours to read outside class.

We will walk through them together only if there is time left today. Nothing in **a01** asks you to write a makefile, so treat this as background for the projects later in the semester.

Building with `make`

Three files, one command. `make` recompiles only what changed.

```
CXX      = g++
CXXFLAGS = -Wall -std=c++17

game: main.o dice.o
  $(CXX) $(CXXFLAGS) -o game main.o dice.o

main.o: main.cpp dice.h
  $(CXX) $(CXXFLAGS) -c main.cpp

dice.o: dice.cpp dice.h
  $(CXX) $(CXXFLAGS) -c dice.cpp

clean:
  rm -f game *.o
```

> Those indents must be **real tabs**, not spaces, the one rule that bites everyone once.

Why `make` Earns Its Keep

- A target is **rebuilt only if** something it depends on is newer
- Change `dice.cpp` → only `dice.o` recompiles, then it relinks
- `make clean` throws away everything built, so you can start fresh
- Your whole build recipe lives in the repo, not in someone's memory

> On a three-file project this saves seconds. On a big one it saves your afternoon, and it's how the autograder builds your code.

Week 01 Recap

- `if`/`switch` and loops read the same as Python, the braces and types are new
- A function must be **declared before it is called**: that's what headers are for
- Arrays are passed as **addresses**, so functions need the size too
- Choose the direction (in / out / in/out) first, then the mechanism
- *Extra:* `make` rebuilds only what changed, and tabs matter

Next: a01 →

This deck stands on earlier CS112 materials by
Joel Adams and **Victor Norman** · adapted and extended by **Eric Araújo**