

6.S096

Introduction to C and C++

Why?

1

You seek performance

1

You seek performance

“zero-overhead principle”

2

You seek to interface
directly with hardware

3

That's kinda it

C

a nice way to avoid writing
assembly language directly

C++

responds to the demands of
maintaining large C projects

C++11

responds to the demands of
maintaining large C++ projects

Maintain power and flexibility
of what came before

Today:
**compilation
Pipeline**

Source Code



Program

main.c

prog

int a = 1;

@*)!%

./prog

gcc -o prog main.c

main.c

```
int a = 1;
```

main2.c

```
int b = 2;
```

prog

```
@*)!%
```

→ ./prog

```
gcc -o prog main.c main2.c
```

```
$ gcc -o prog main.c
$ ./prog
Hello, World!
$
```

```
$ gcc -o prog main.c  
$ ./prog  
Hello, World!
```

\$

DONE

“To debug the sausage, one
must see how it is made.”

—Someone, probably

Main.java

```
int a = 1;
```

Main.class

```
%!(*@
```

java Main



Main.java

```
int a = 1;
```

Main.class

```
%!(*@
```

main.py

```
a = 1
```

main.pyc

```
%!(*@
```

java Main

python main.py

Main.java

```
int a = 1;
```

Main.class

```
%!(*@
```

main.py

```
a = 1
```

main.pyc

```
%!(*@
```

java Main

main.c

```
int a = 1;
```

```
int a = 1;
```

main.o

```
%!(*@
```

prog

```
@*)!%
```

python main.py

```
%!(*@
```

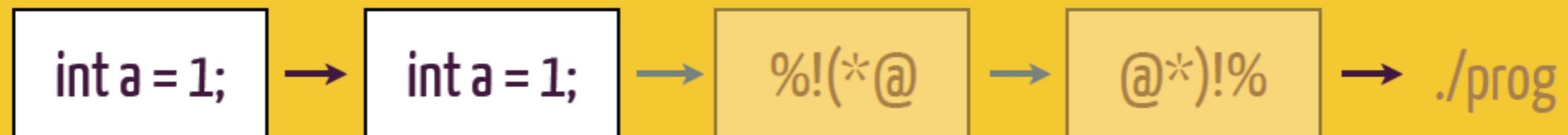
```
@*)!%
```

.lprog

main.c

main.o

prog



Pre-Process

main.c

int a = 1;

main.o

int a = 1;

prog

%!(*@

@*)!%

./prog

Pre-Process

Compile

main.c

main.o

prog

int a = 1;

int a = 1;

%!(*@

@*)!%

./prog

Pre-Process

Compile

Link

main.c

int a = 1;

int a = 1;

main.o

prog

%!(*@

@*)!%

./prog

Pre-Process

main2.o

%!(*@

Compile

Link

Pre-Process

Compile

Link

Pre-Process

Compile

Link

#include

#define

#ifdef

rimshot.txt

```
ba-dum chh
```

joke.txt

```
A man walks into  
a bar. Ouch!  
#include "rimshot.txt"
```

cpp -P joke.txt

output:

```
A man walks into  
a bar. Ouch!  
ba-dum chh
```

```
cpp -P joke.txt
```

double.py

```
#define fosho def  
#define kthx return  
#define wutz print
```

```
fosho double(x):  
    kthx x * 2  
wutz double(6)
```

These are called
“macros”

cpp -P double.py

output:

```
def double(x):  
    return x * 2  
print double(6)
```

cpp -P double.py

output:

```
def double(x):  
    return x * 2  
print double(6)
```

cpp -P double.py | python

12

AWESOME

cpp -P double.py

beer.txt

```
#define beer(x) x bottles of \
beer on the wall...
```

```
beer(99)
beer(98)
beer(97)
```

```
...
```

```
cpp -P beer.txt
```

beer.txt

```
#define beer(x) x bottles of \
beer on the wall...
```

```
beer(99)
beer(98)
beer(97)
```

```
...
```

```
cpp -P beer.txt
```

output:

```
99 bottles of beer on the wall...
98 bottles of beer on the wall...
97 bottles of beer on the wall...
...
...
```

cpp -P beer.txt

answer.txt

```
What's 7 times 6?
```

```
#ifdef REVEAL
```

```
42
```

```
#endif
```

```
cpp -P answer.txt
```

output:

What's 7 times 6?

cpp -P answer.txt

output:

What's 7 times 6?



cpp -P answer.txt

```
#define REVEAL
```

```
cpp -P answer.txt
```

```
#define REVEAL
```

or:

```
cpp -P -D REVEAL answer.txt
```

```
cpp -P answer.txt
```

output:

What's 7 times 6?

42

cpp -P -D REVEAL answer.txt

answer.txt

What's 7 times 6?

```
#ifndef REVEAL
```

42

```
#endif
```

cpp -P answer.txt

(Fancy) String Substitution

How is this used in C?

hello.c

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

gcc -E hello.c

hello.c

* angle brackets -> use the system search path

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

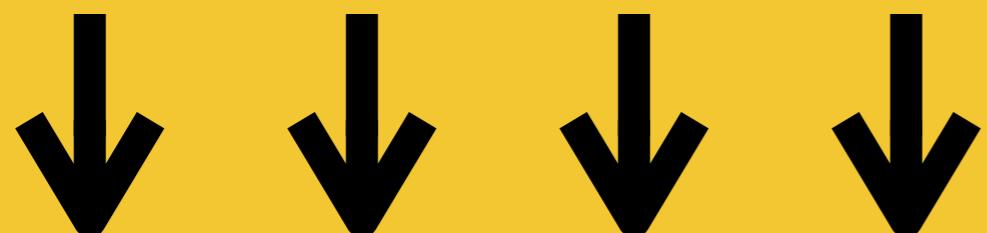
gcc -E hello.c

hello.c

* angle brackets -> use the system search path

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```



gcc -E hello.c

output:

```
int printf(const char * , ...)  
__attribute__((format_  
(printf_, 1, 2)));  
  
int main() {  
    printf("Hello, World!\n");  
}
```

* pretending printf is all that's defined in stdio.h

gcc -E hello.c

output:

```
int printf(const char * , ...);  
  
int main() {  
    printf("Hello, World!\n");  
}
```

* pretending printf is all that's defined in stdio.h

gcc -E hello.c

#include is not

```
import pickle
```

```
import java.io.*;
```

fib.c

```
#define MAX_FIB 20
int fib[MAX_FIB];

int main() {
    fib[0] = 0;
    fib[1] = 1;
    for(int i = 2; i < MAX_FIB; i++)
        fib[i] = fib[i-1] + fib[i-2];
    return 0;
}
```

gcc -E fib.c

output:

```
int fib[20];

int main() {
    fib[0] = 0;
    fib[1] = 1;
    for(int i = 2; i < 20; i++)
        fib[i] = fib[i-1] + fib[i-2];
}
```

gcc -E fib.c

debug.c

```
#include <stdio.h>

int main() {
#ifndef DEBUG
    printf("Hello, World!\n");
#endif
    return 0;
}
```

```
gcc -DDEBUG debug.c -o debug
```

debug.c

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
gcc -DDEBUG debug.c -o debug
```

debug.c

```
#include <stdio.h>

int main() {
    return 0;
}
```

```
gcc debug.c -o debug
```

Pre-Process

Compile

Link

main.c

main.o

prog

int a = 1;

int a = 1;

%!(*@

@*)!%

./prog

Compile

Compile

Type-checking
Linear processing

Type-checking

```
int reptile() {  
    return "frog";  
}
```

Type-checking

```
int reptile() {  
    return "frog";  
}
```

reptile.c: In function ‘reptile’:
reptile.c:2:5: warning: return makes
integer from pointer without a cast

Type-checking

```
def vegetable(day):  
    if day != "Tuesday":  
        return "tomato"  
    else:  
        return 1000
```

Type-checking

```
def vegetable(day):  
    if day != "Tuesday":  
        return "tomato"  
    else:  
        return 1000
```

Python says: no problem

```
int reptile() {  
    return "frog";  
}
```

```
int () {  
    return char*;  
}
```

```
int vegetable(char *day) {  
    if (strcmp(day, "Tuesday") != 0){  
        return "tomato";  
    } else {  
        return 1000;  
    }  
}
```

```
int (char*) {
    if (int){
        return char*;
    } else {
        return int;
    }
}
```



```
int (char*) {  
    if (int){  
        return char*;  
    } else {  
        return int;  
    }  
}
```

```
int (char*) {
    if (int){
        return char*; ←
    } else {
        return int;
    }
}
```

Everything has
a single, fixed type

```
def foo(a, b):  
    return a + b  
  
foo(2, 3)  
foo("2", "3")
```

Variable Declarations

```
int foo;  
float foo;  
double foo;  
char foo;
```

```
int foo[42];  
int *foo;  
struct Bar foo;
```

Function Declarations

```
double fmin(double, double);
```



return type



argument types



Function Declarations

```
void exit(int);
```



returns nothing

```
int rand(void);
```



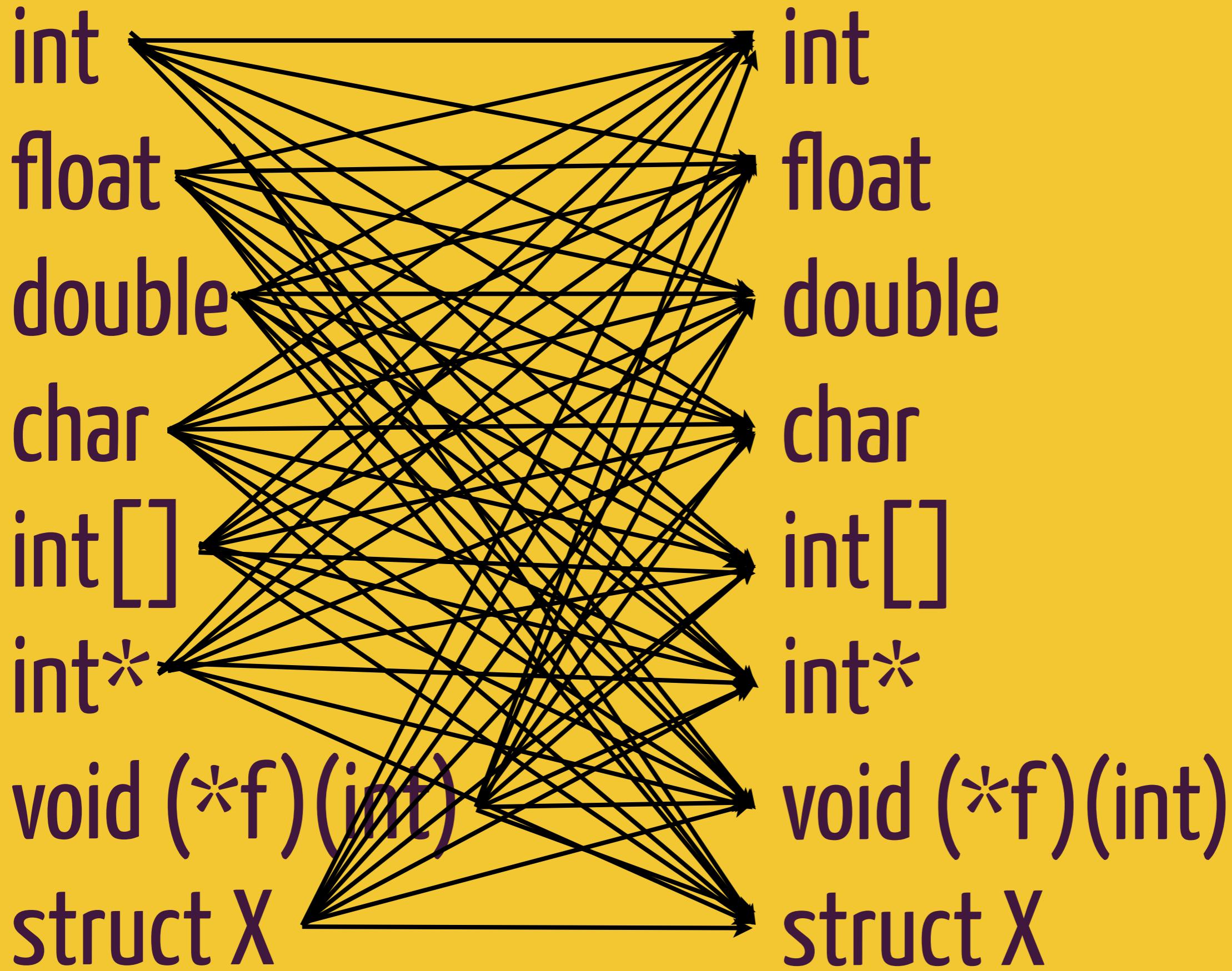
takes no arguments

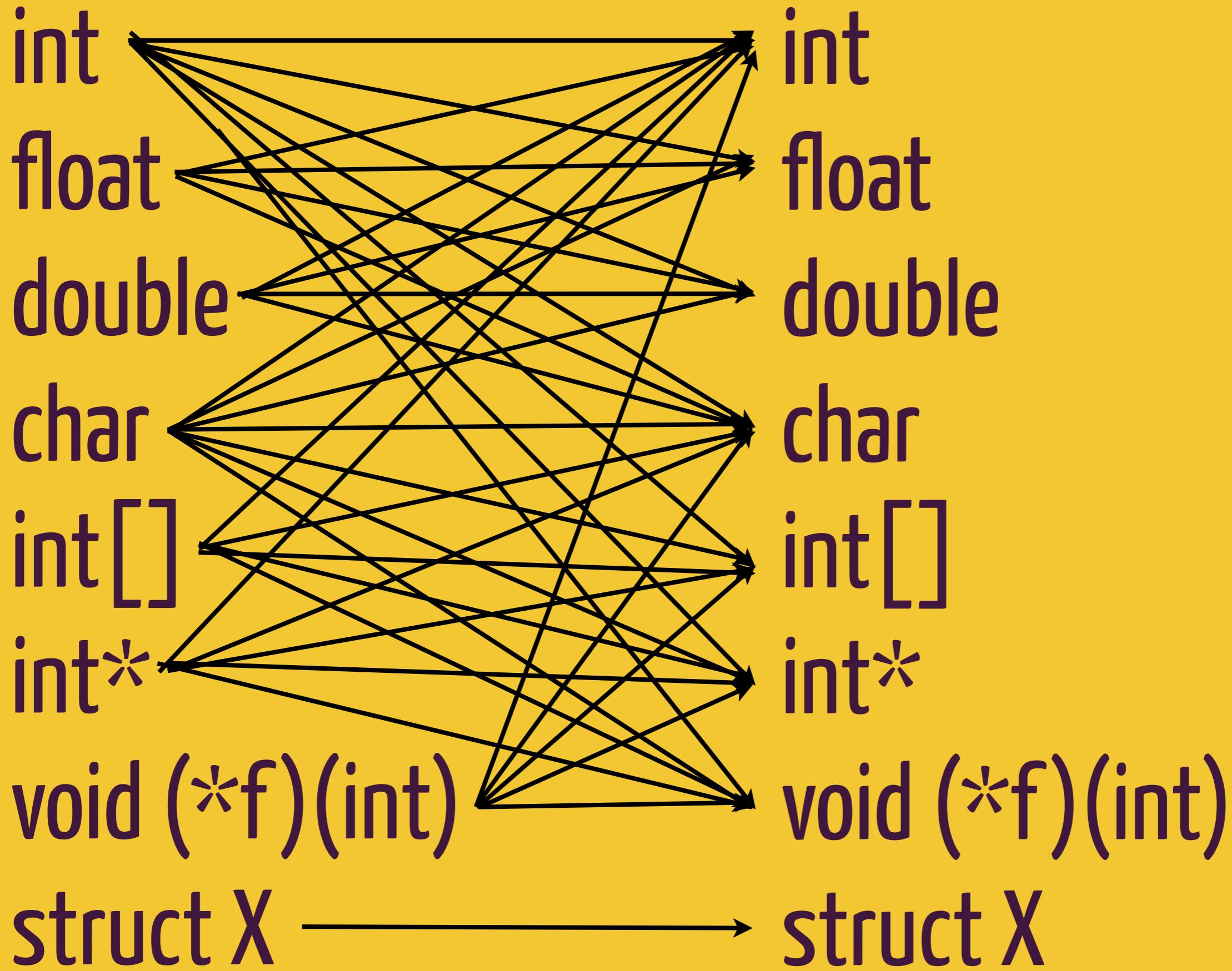
```
int foo(int a, int b){  
    return a + b;  
}
```

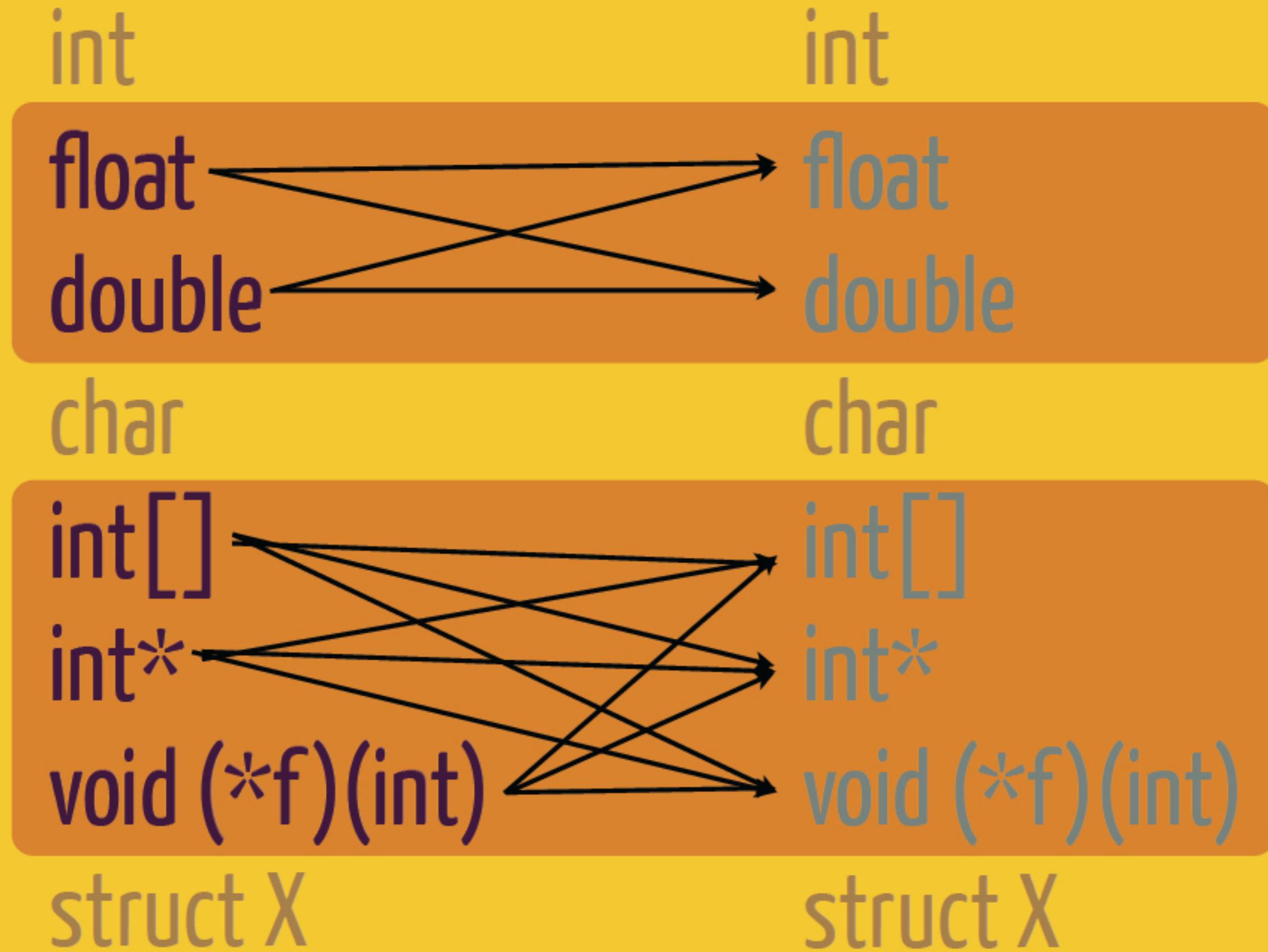
reptile.c: In function ‘reptile’:
reptile.c:2:5: warning: return makes
integer from pointer without a cast

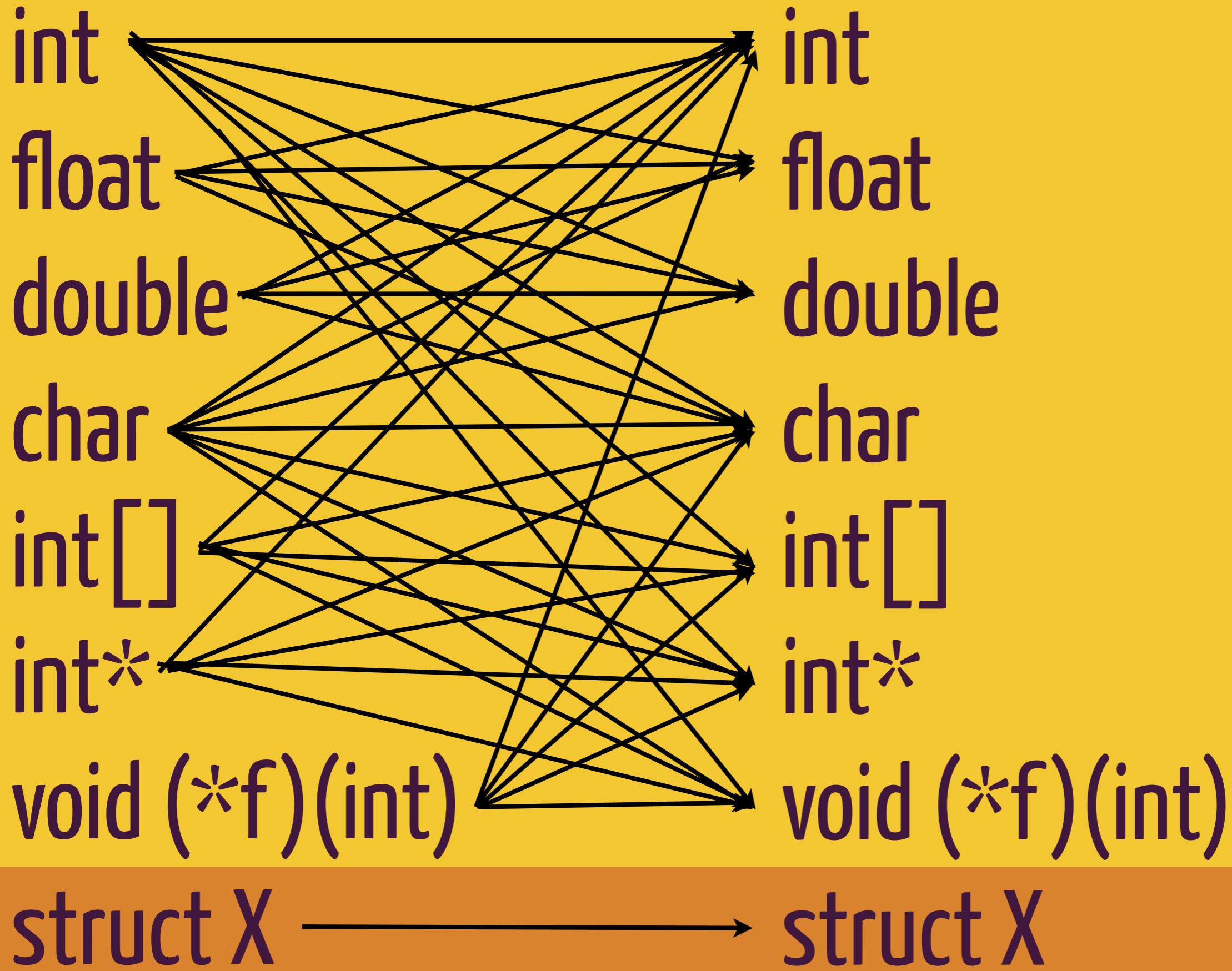
reptile.c: In function ‘reptile’:
reptile.c:2:5: warning: return makes
integer from pointer without a cast

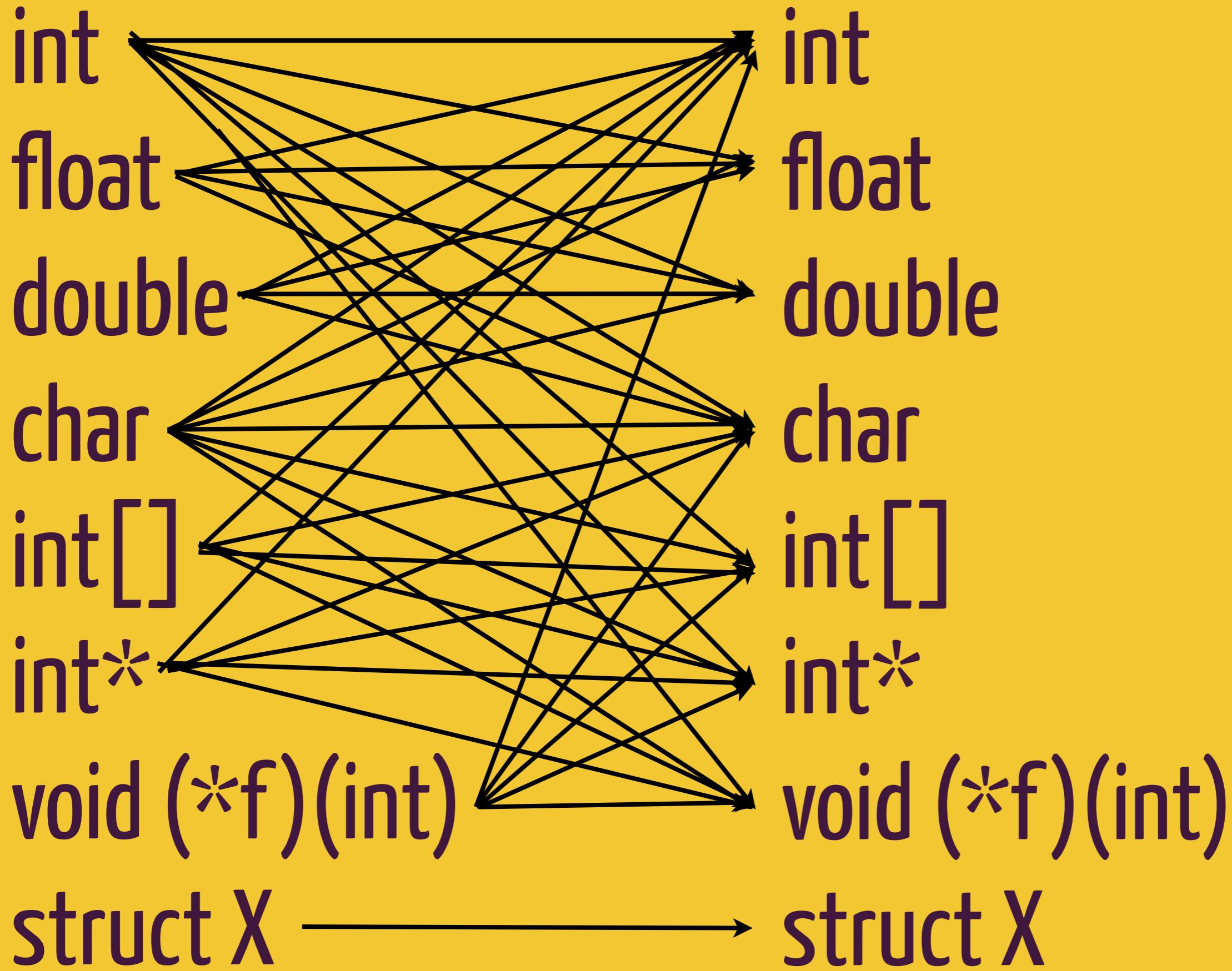
```
int a = 4;  
float b = (float)a;
```

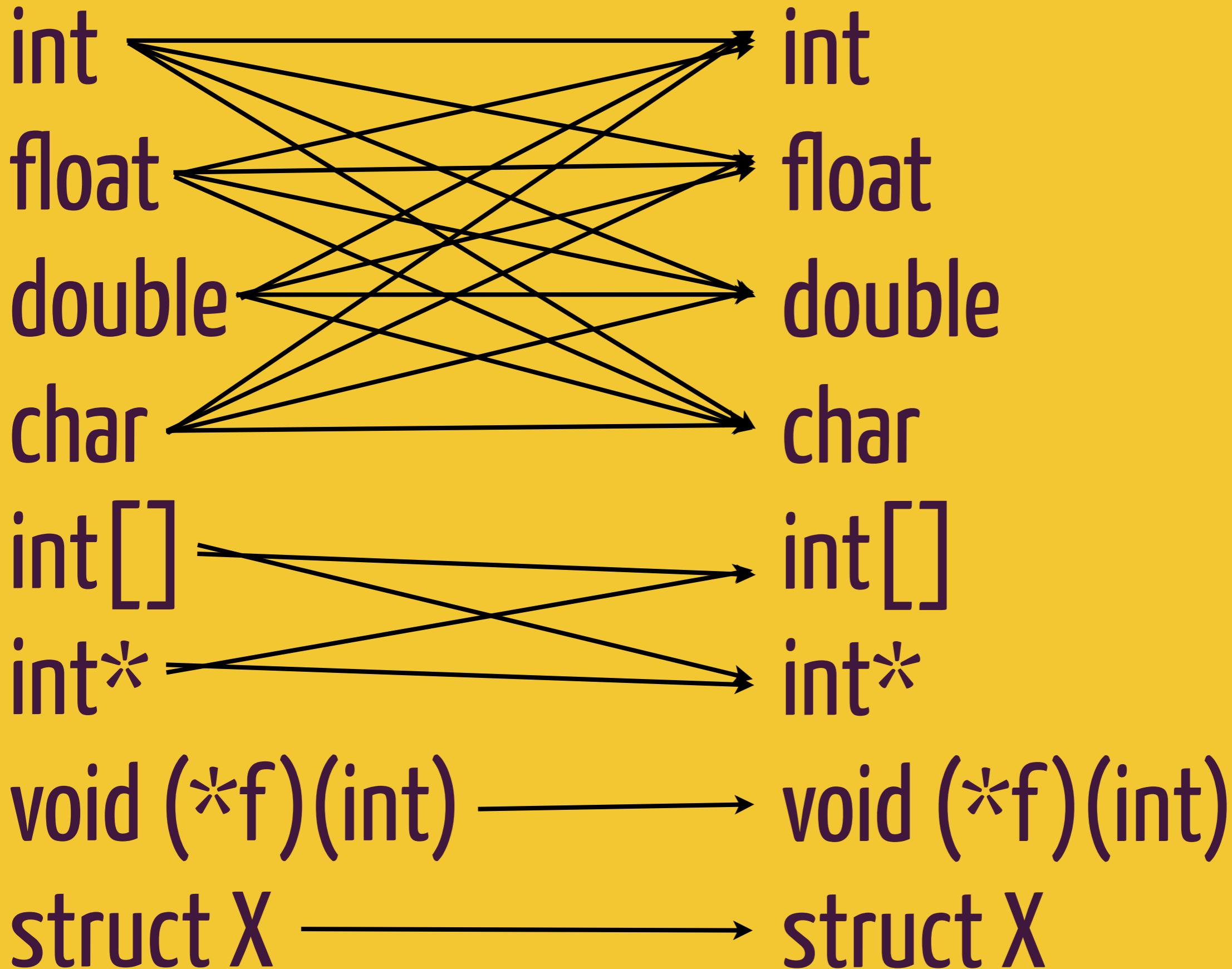


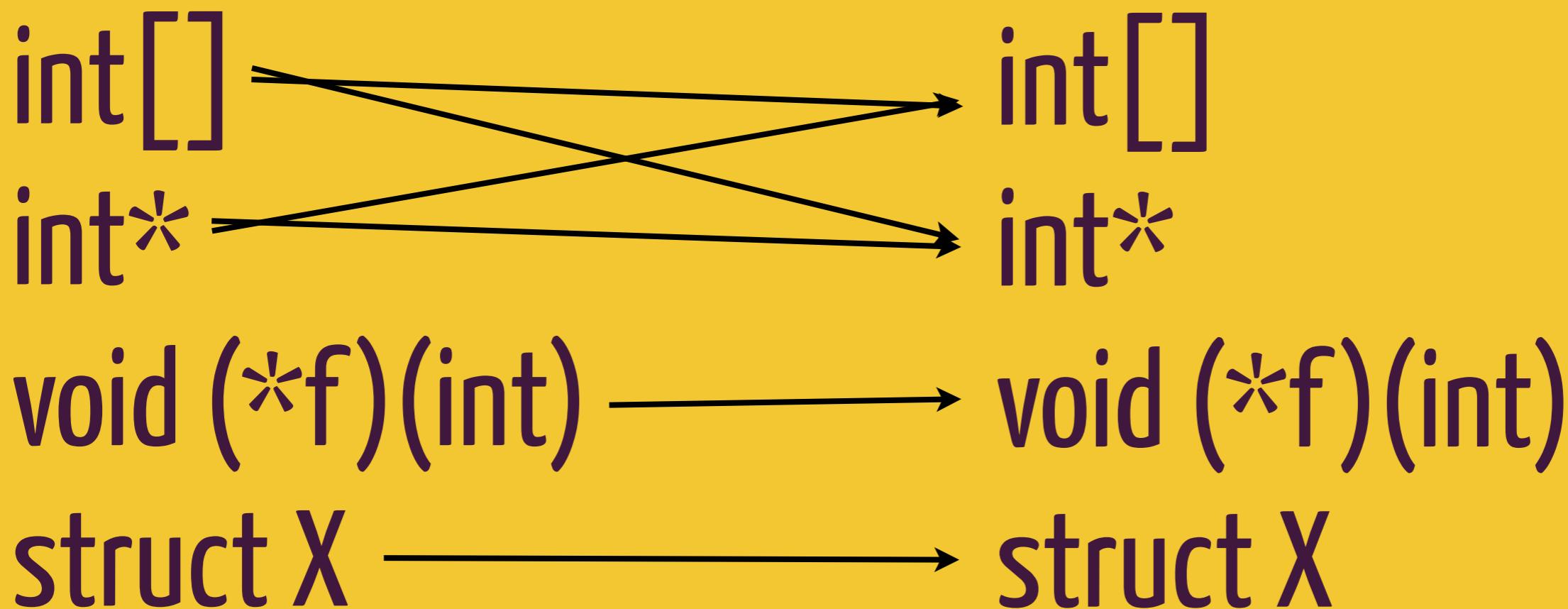
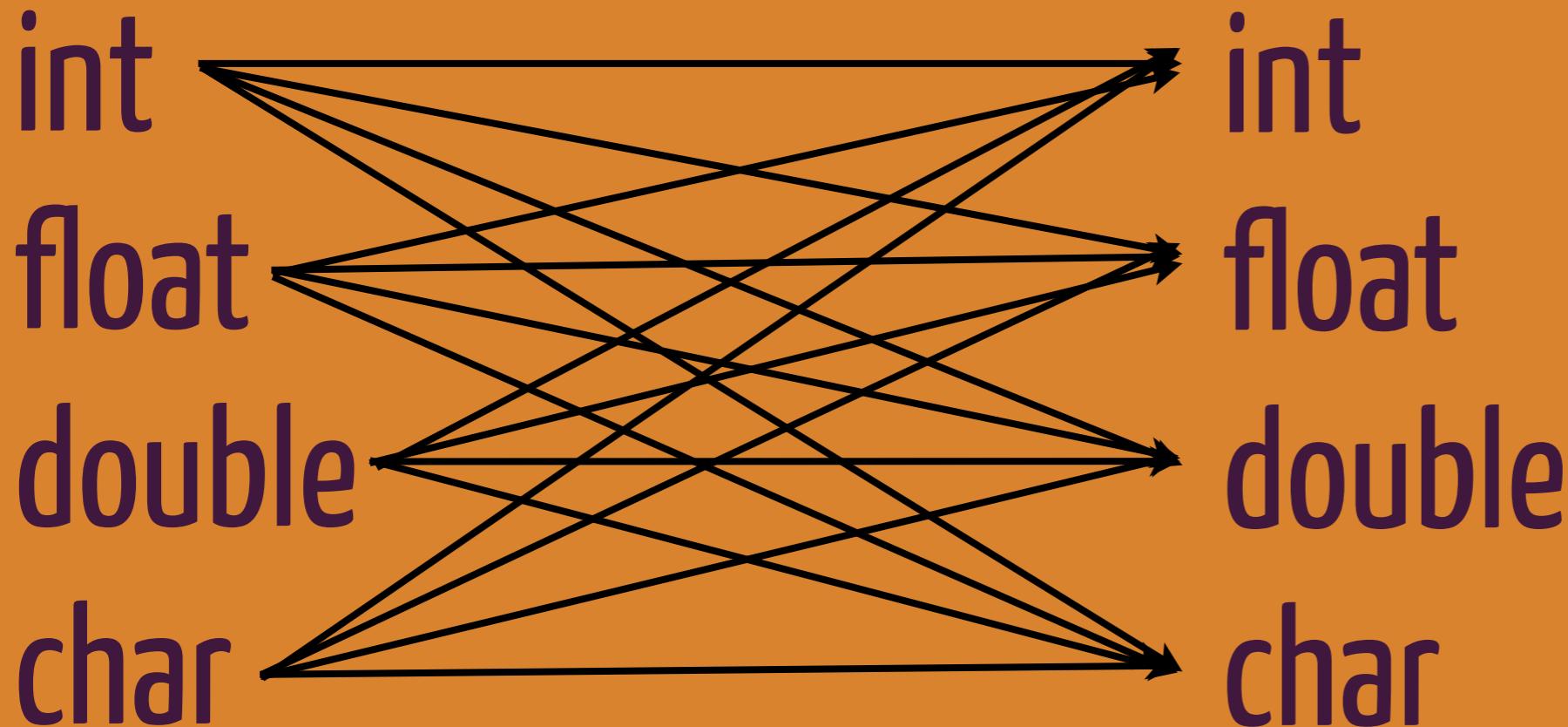


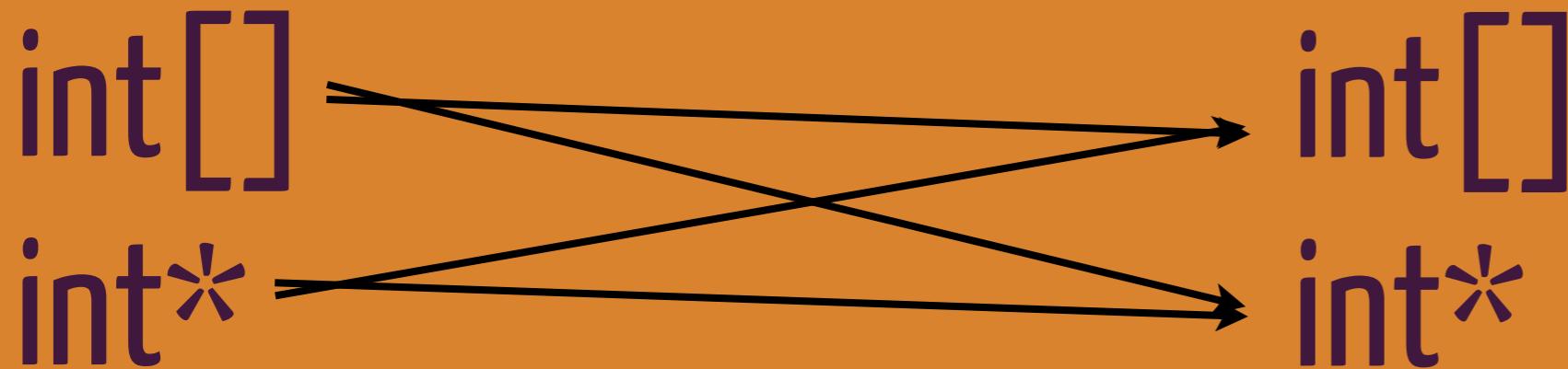
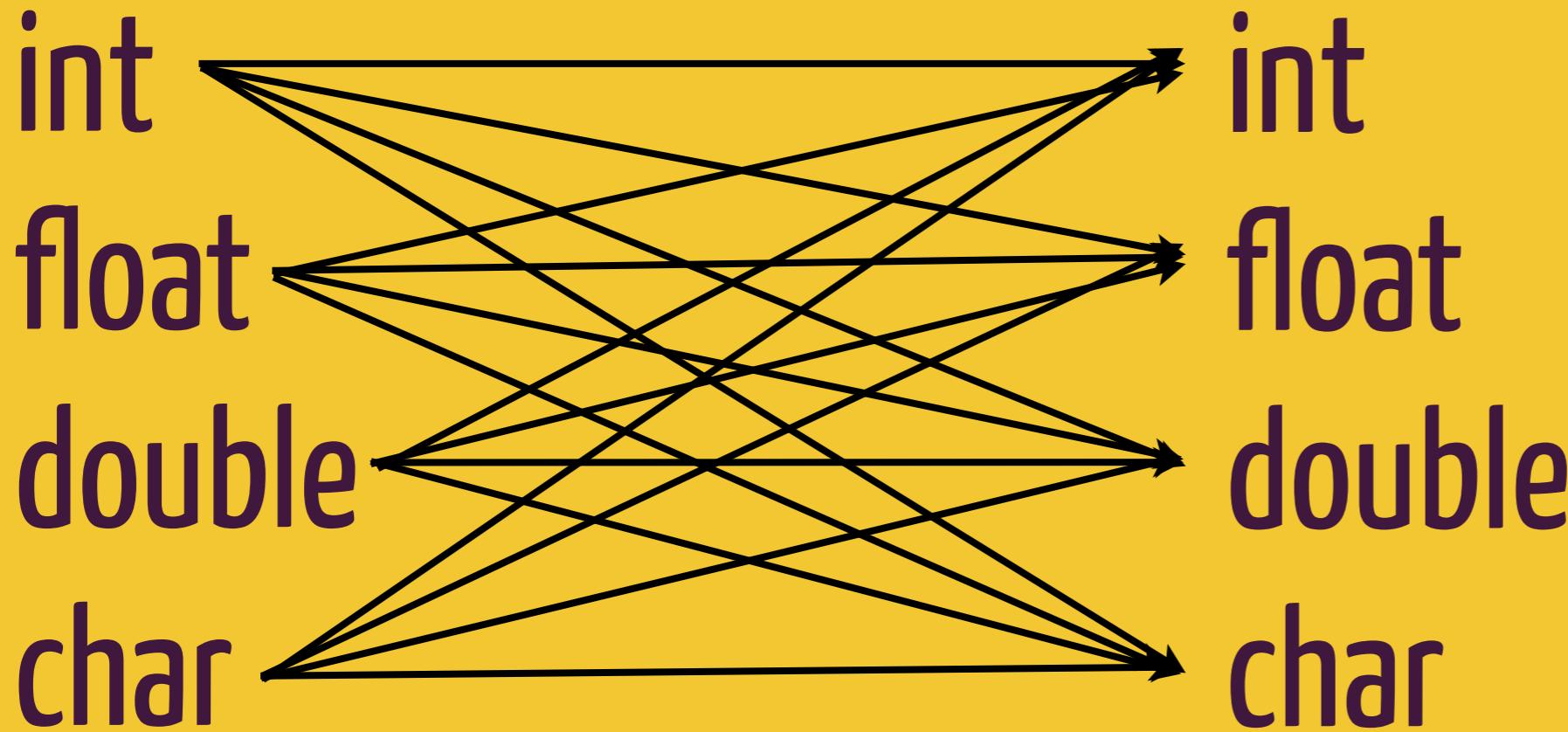






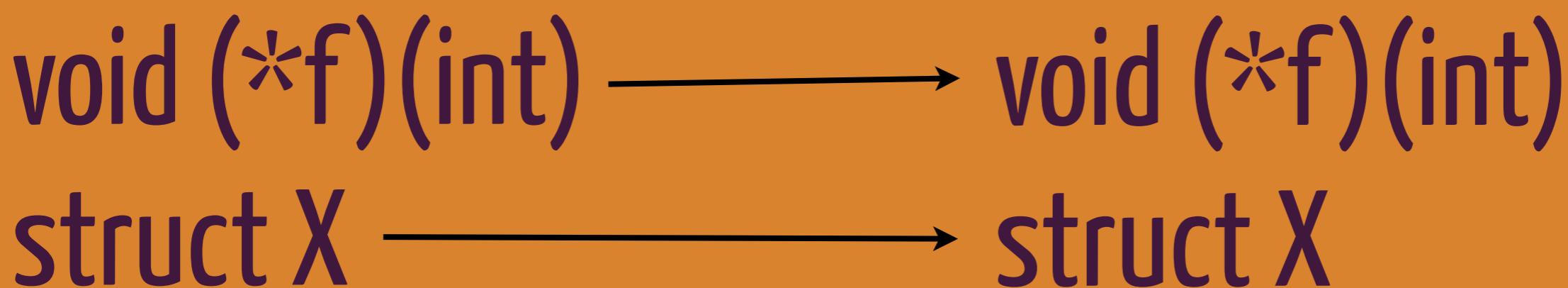
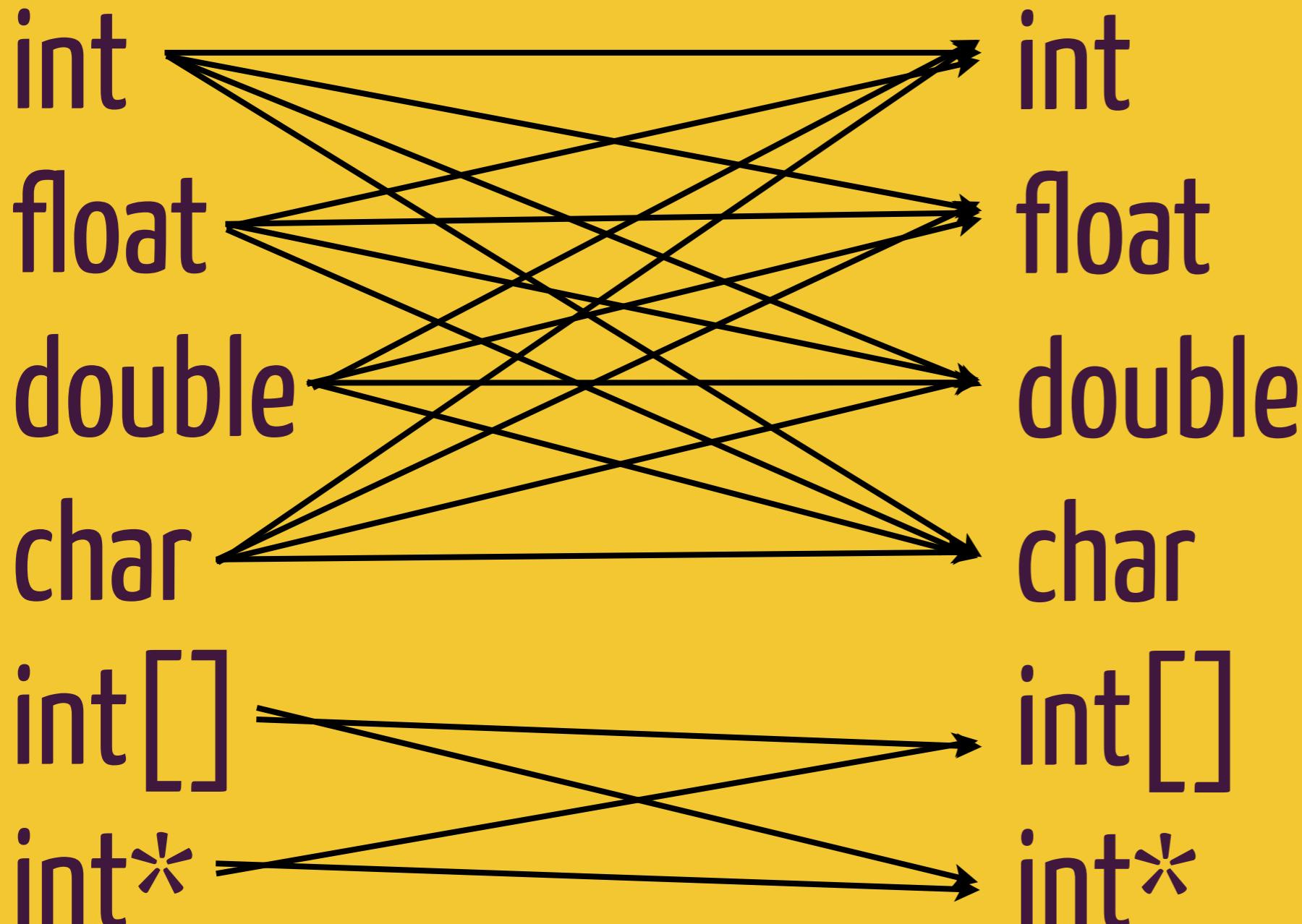






void (*f)(int) —————→ void (*f)(int)

struct X —————→ struct X



Compile

Type-checking
Linear processing

Linear processing

(just a small note)



You can only use
what's declared above

```
int main() {  
    printf("%d\n", answer() );  
    return 0;  
}
```



```
int answer() {  
    return 1337;  
}
```

```
int main() {  
    printf("%d\n", answer());  
    return 0;  
}  
  
int answer() {  
    return 1337;  
}
```

```
int answer() {  
    return 1337;  
}  
  
int main() {  
    printf("%d\n", answer());  
    return 0;  
}
```

```
int answer(); ← declaration  
int main() {  
    printf("%d\n", answer());  
    return 0;  
}  
  
int answer() { ← definition  
    return 1337;  
}
```

HIMMAM

```
int answer(); ← declaration  
int main() {  
    printf("%d\n", answer());  
    return 0;  
}  
  
int answer() { ← definition  
    return 1337;  
}
```

```
int answer(); ← declaration  
int main() {  
    printf("%d\n", answer());  
    return 0;  
}
```

```
int answer() { ← definition  
    return 1337;  
}
```

```
int answer(); ← declaration
```

```
#include "answer.h"
```

```
int main() {  
    printf("%d\n", answer() );  
    return 0;  
}
```

```
int answer() { ← definition  
    return 1337;  
}
```

Pre-Process

Compile

Link

main.o

```
int main()
```

```
int answer()
```



main.o

```
int main()
```

```
answer.c: In function ‘main’:  
answer.c:4: warning: implicit declaration of  
function ‘answer’  
Undefined symbols for architecture x86_64:  
" _answer", referenced from:  
      _main in ccuzmRrm.o  
ld: symbol(s) not found for architecture x86_64  
collect2: ld returned 1 exit status
```

answer.c: In function ‘main’:
answer.c:4: warning: implicit declaration of
function ‘answer’

Undefined symbols for architecture x86_64:
" _answer", referenced from:
 _main in ccuzmRrm.o
ld: symbol(s) not found for architecture x86_64
collect2: ld returned 1 exit status

```
answer.c: In function ‘main’:
answer.c:4: warning: implicit declaration of
function ‘answer’
```

```
Undefined symbols for architecture x86_64:
        "_answer", referenced from:
                _main in ccuzmRrm.o
ld: symbol(s) not found for architecture x86_64
collect2: ld returned 1 exit status
```

Compiler: “I don’t know what answer is.
I’ll assume it returns an int.”

```
answer.c: In function ‘main’:  
answer.c:4: warning: implicit declaration of  
function ‘answer’
```

```
Undefined symbols for architecture x86_64:  
“_answer”, referenced from:  
    _main in ccuzmRrm.o  
ld: symbol(s) not found for architecture x86_64  
collect2: ld returned 1 exit status
```

```
answer.c: In function ‘main’:  
answer.c:4: warning: implicit declaration of  
function ‘answer’
```

```
Undefined symbols for architecture x86_64:  
“_answer”, referenced from:  
    _main in ccuzmRrm.o  
ld: symbol(s) not found for architecture x86_64  
collect2: ld returned 1 exit status
```

Linker: “I looked in all the object files,
but I couldn’t find answer.”

main.o

```
int main()
```

main.o

answer.o

```
int main()
```

```
int answer()
```

main.o

```
int main()
```

answer.o

```
int answer()
```

```
int main() {  
    printf("%d\n", answer());  
    return 0;  
}
```

```
int answer() {  
    return 1337;  
}
```

main.o

answer.o

```
int main()
```

```
int answer()
```

prog

```
int main()
```

```
int answer()
```

gcc -o prog main.c answer.c

answer.c: In function ‘main’:
answer.c:4: warning: implicit declaration of
function ‘answer’

```
answer.c: In function ‘main’:  
answer.c:4: warning: implicit declaration of  
function ‘answer’
```

Compiler: “I don’t know what answer is.
I’ll assume it returns an int.”

answer.h

```
int answer();
```

main.c

```
#include "answer.h"

int main() {
    printf("%d\n", answer());
    return 0;
}
```

answer.h

```
int answer();
```

answer.c

```
#include "answer.h"

int answer() {
    return 1337;
}
```

Summary

answer.h

```
int answer();
```

main.c

```
#include "answer.h"

int main() {
    printf("%d\n", answer());
    return 0;
}
```

Preprocess: gcc -E main.c

```
int answer();

int main() {
    printf("%d\n", answer());
    return 0;
}
```

Compile: gcc -c main.c main.c

main.o

```
% ! (*@
```

answer.o

```
% ! (*@
```

Link: gcc -o prog main.o main.o

prog

```
% ! (*@
```

Pre-Process

Compile

Link

MIT OpenCourseWare
<http://ocw.mit.edu>

6.S096 Introduction to C and C++
IAP 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.



MIT OpenCourseWare

6.S096 | January IAP 2013 | Undergraduate

Introduction To C And C++

≡ Menu

More Info

Feedback

Lectures and Assignments

Compilation Pipeline

Lecture Notes

[Lecture 1: Compilation Pipeline \(PDF - 1.0MB\)](#)

Lab Exercises

The primary goal of this lab period is to get your C compiler up and running.

We have 2 “Hello, World!” examples for you to practice compiling to make sure that everything is working.

[Lab 1 files \(ZIP\)](#) (This ZIP file contains: 3 .c files and 1 .h file.)

Compile `hello1` with:

```
gcc hello.c -o hello1
```

Compile `hello2` with:

```
gcc main.c hello.c -o hello2
```

Assignment 1

Setup

[Assignment 1 files \(ZIP\)](#) (This ZIP file contains: 3 .c files and 2 .h files.)

The zip contains 3 C files:

1. fibeverse.c
2. fibonacci.c
3. reverse.c

And 2 header files (.h):

1. fibonacci.h
2. reverse.h

You can compile them with this command (though it won't work at first; see Problem 1):

```
gcc -Wall -std=c99 _fibeverse.c reverse.c fibonacci.c -o **fibeverse**
```

You can run the resulting program with two arguments: a number, then a string (in quotes):

```
./fibeverse 6 'what a trip that was!' 8 was! that trip a what
```

The first line it prints is the 6th fibonacci number. The second line is the string you provided, with the words reversed.

Problem 1

Unfortunately, the code doesn't compile as-is! Fix the compile errors and warnings. `gcc` should produce no output with the above command when you are done.

Problem 2

I can't decide whether I want a program that computes Fibonacci numbers or a program that reverses strings! Let's modify fibeverse so that it can be compiled into either.

Use the preprocessor macros we taught in class to make it so that I can choose which program it is at compile time.

When I compile it with this command, it should compute the Fibonacci number but not reverse the second argument:

```
gcc -Wall -std=c99 **-DFIBONACCI** fibeverse.c reverse.c fibonacci.c -o
**fibonacci**
```

Then I can run it like this:

```
./fibonacci 8
```

When I use this command, it should reverse the string I provide as the first argument, and not do any fibonacci calculation:

```
gcc -Wall -std=c99 **-DREVERSE** fibeverse.c reverse.c fibonacci.c -o **reverse**
```

Then I can run it like this:

```
./reverse 'a brave new world'
```

It should work as it originally did when I provide both compiler flags:

```
gcc -Wall -std=c99 **-DFIBONACCI -DREVERSE** fibeverse.c reverse.c fibonacci.c -o
**fibeverse**
```

Solutions

[Assignment 1 solution \(PDF\)](#)



Over 2,500 courses & materials

Freely sharing knowledge with learners and educators around the world. [Learn more](#)

[Accessibility](#)

[Creative Commons License](#)

[Terms and Conditions](#)

Proud member of: The logo for Open Education GLOBAL, featuring a stylized globe icon and the text "Open Education GLOBAL".



© 2001–2025 Massachusetts Institute of Technology

6.s096

Lecture 2

Today

The core of the language

- Control Structures
- Variables and Functions
- Scope
- Uninitialized Memory - and what to do about it!

Control Structures

Basic Control Structures

You've probably seen these...

while

do...while

for

if [...else if], [...else]

```
int i = 0;  
while(i++ < 3){  
    printf("%d ", i);  
}
```

=> 1 2 3

Basic Control Structures

You've probably seen these...

while

do...while

for

if [...else if], [...else]

```
int i = 0;  
do {  
    printf("%d ", i);  
} while(i++ < 3);
```

=> 0 1 2 3

Basic Control Structures

You've probably seen these...

while

do...while

for

if [...else if], [...else]

```
// C99-style
for(int i = 0; i < 3; ++i){
    printf("%d ", i);
}
```

=> 0 1 2

Basic Control Structures

while

do...while

for

if [...else if], [...else]

You've probably seen these...

```
int i = 0;  
if(i < 3){  
    printf("It sure is.");  
} else if(i == 3){  
    printf("Nope.");  
} else {  
    printf("Still nope.");  
}
```

Slight variations

- Blocks / braces often optional (if, while, for):
`if(condition) expression;`
- Empty for loop is an “infinite” while:
`for(;;) expression;`

switch

- ```
switch(i){
 case 1:
 printf("It's one!");
 break;
 case 2:
 printf("It's two!!");
 break;
 default:
 printf("It's something else!!!");
}
```

# Jumps

Output:

0 1 2 4 5 6 the end

```
void foo(){
 for(int i = 0; i < 10; ++i){
 printf("%d ", i);
 if(i == 2){
 i = 3;
 continue;
 } else if(i == 6){
 break;
 }
 }
 goto end;
 printf("near the end\n");
end:
 printf("the end\n");
 return;
 printf("or is it?");
}
```

# Jumps

Output:

0 1 2 4 5 6 the end

```
void foo(){
 for(int i = 0; i < 10; ++i){
 printf("%d ", i);
 if(i == 2){
 i = 3;
 continue;
 } else if(i == 6){
 break;
 }
 }
 goto end;
 printf("near the end\n");
end:
 printf("the end\n");
 return;
 printf("or is it?");
}
```



# Jumps

Output:

0 1 2 4 5 6 the end

```
void foo(){
 for(int i = 0; i < 10; ++i){
 printf("%d ", i);
 if(i == 2){
 i = 3;
 continue;
 } else if(i == 6){
 break;
 }
 }
 goto end;
 printf("near the end\n");
end:
 printf("the end\n");
 return;
 printf("or is it?");
}
```

# Jumps

Output:

0 1 2 4 5 6 the end

```
void foo(){
 for(int i = 0; i < 10; ++i){
 printf("%d ", i);
 if(i == 2){
 i = 3;
 continue;
 } else if(i == 6){
 break;
 }
 }
 goto end;
 printf("near the end\n");
end:
 printf("the end\n");
 return;
 printf("or is it?");
}
```

# Jumps

## Output:

0 1 2 4 5 6 the end

```
int main(int argc, char ** argv){
 foo();
 ←return 0;
}
```

```
void foo(){
 for(int i = 0; i < 10; ++i){
 printf("%d ", i);
 if(i == 2){
 i = 3;
 continue;
 } else if(i == 6){
 break;
 }
 }

 goto end;
 printf("near the end\n");
end:
 printf("the end\n");
 return;
 printf("or is it?");
}
```

# The goto statement in detail

- Syntax:

`goto Label;`

... where *label* refers to an earlier or later labelled section of code.

- Target label must be in the same function as the goto statement.
- Notorious for creating hard-to-read code, but the concept is critical to how computers operate.

# Variables and Functions

# Variables and constants

```
int a = 1; const int b = 1;
a = 2; // cool b = 2;
// error:
// read-only variable
// is not assignable
```

# static Variables

Static variables retain their value throughout the life of the program.

```
void foo(){
 static int count = 0;
 printf("%d ", count++);
}
```

```
for(int i = 0; i < 5; ++i){
 foo();
}
```

Output: 0 1 2 3 4

# Functions in Variables

We'll examine part of this syntax in more depth in later lectures.

```
int foo(int a, int b){
 return a + b;
}
```

```
int (*func)(int, int) = &foo;
int result = func(2, 2);
printf("%d ", result); // 4
```

```
int bar(int c, int d){
 return c - d;
}
```

```
func = &bar;
result = func(2, 2);
printf("%d", result); // 0
```

# Scope

# Scope

A variable has a *scope* in which it is said to be defined.

```
void bar(){
 int a = 0;
 if(3 > 0){
 int b = 0;
 b = 2; // okay
 }
 a++; // okay
 b++; // error:
 // use of undeclared
 // identifier 'b'
}
```

```
void foo(){
 int a = 0;
}
```

In **foo** and **bar**,  
**a** is “in scope” for  
the entire function.  
**b** is “in scope” only within  
the if statement’s block in **bar**.

# Anonymous Blocks

Anonymous blocks demonstrate the concept of *block scope*.

```
void foo(){
 { int a = 0; }
 {
 double a = 3.14; // no problem!
 {
 char * a = "3.14"; // no problem!
 }
 }
 // no 'a' defined in this scope
}
```

# Uninitialized Memory

When you see that gibberish output...

# Program memory, simplified...

```
int a = 0;
```

```
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
```

(Common)

# Sources

Avoid these situations if you can help it!

- Uninitialized variables:

```
int i;
printf("%d", i);
```

- Out-of-bounds array access:

```
char reversed[20];
char out_of_bounds = reversed[21];
```

- Variables passed out of their defining function's scope.

- `malloc` (coming up in a later lecture)

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.S096 Introduction to C and C++  
IAP 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.



MIT OpenCourseWare

6.S096 | January IAP 2013 | Undergraduate

# Introduction To C And C++

≡ Menu

More Info

Feedback

## Lectures and Assignments

### Core C: Control Structures, Variables, Scope, and Uninitialized Memory

#### Lecture Notes

[Lecture 2: Core C \(PDF\)](#)

#### Lab Exercises

The primary goal of this lab period is to make sure you know all of the basics of writing code in C and to avoid common pitfalls unique to C and C++.

##### Exercise 1

Starting with an empty .c file, write a C program to calculate the first 100 [triangular numbers](#) (0, 1, 3, 6, 10, 15, 21, ...). You may check back at previous example code and slides if you get stuck, but take this an opportunity to try to recall, from memory, the standard boilerplate code you'll be writing often when starting a new C program, like the proper `#include` statements to be able to print to the console.

##### Exercise 2

Find and fix the variable shadowing bug in the following C program, which is intended to calculate the factorial function:

```
#include <stdio.h>

int factorial (int n) {
 int i = 1;
 while (n > 1) {
 i = i * n;
 int n = n - 1;
 }
 return i;
}

int main (int argc, char *argv[]) {
 int fac4 = factorial(4);
 int fac5 = factorial(5);
 printf("4! = %d, 5! = %d\n", fac4, fac5);
 return 0;
}
```

## Assignment 2

### Problem 1

Rewrite the program below, replacing the `for` loop with a combination of `goto` and `if` statements. The output of the program should stay the same. That is, the program should print out the arguments passed to it on the command line.

```
#include <stdio.h>

int main(int argc, char ** argv){
 for (int i = 1; i < argc; i++) {
 printf("%s\n", argv[i]);
 }
 return 0;
}
```

If you ran the program like this: `./prog one two three`, it would print

```
one
two
three
```

### Problem 2

In lecture, we covered a variety of control structures for looping, such as `for`, `while`, `do/while`, `goto`, and several for testing conditions, such as `if` and `switch`.

Your task is to find seven different ways to print the odd numbers between 0 and 10 on a single line. You should give them names like `amaze1`, `amaze2`, etc. Here's a bonus amaze (that you can

use as one of your seven if you like):

```
void amazeWOW() {
 int i;
 printf("amazeWOW:\t");
 for (i = 0; i <= 10; i++) {
 if (i % 2 == 1) {
 printf("%d ", i);
 }
 }
 printf("\n");
}
```

You may need to get a little creative!

Put all of your functions in the same C file and call them in order from the `main()` function. We should be able to compile your program with this command and see no errors:

```
gcc -Wall -std=c99 -o amaze amaze.c
```

## Solutions

Solutions are not available for this assignment.



**Over 2,500 courses & materials**

Freely sharing knowledge with learners and educators around the world. [Learn more](#)

[Accessibility](#)

[Creative Commons License](#)

[Terms and Conditions](#)

Proud member of: The logo for Open Education GLOBAL, featuring a stylized blue 'G' icon followed by the text "Open Education GLOBAL".



© 2001–2025 Massachusetts Institute of Technology

# 6.S096: Introduction to C/C++

Frank Li, Tom Lieber, Kyle Murray

## Lecture 3: C Memory Management

January 15, 2013

# Today...

- Computer Memory
- Pointers/Addresses
- Arrays
- Memory Allocation

# Today...

- **Computer Memory**
- Pointers/Addresses
- Arrays
- Memory Allocation

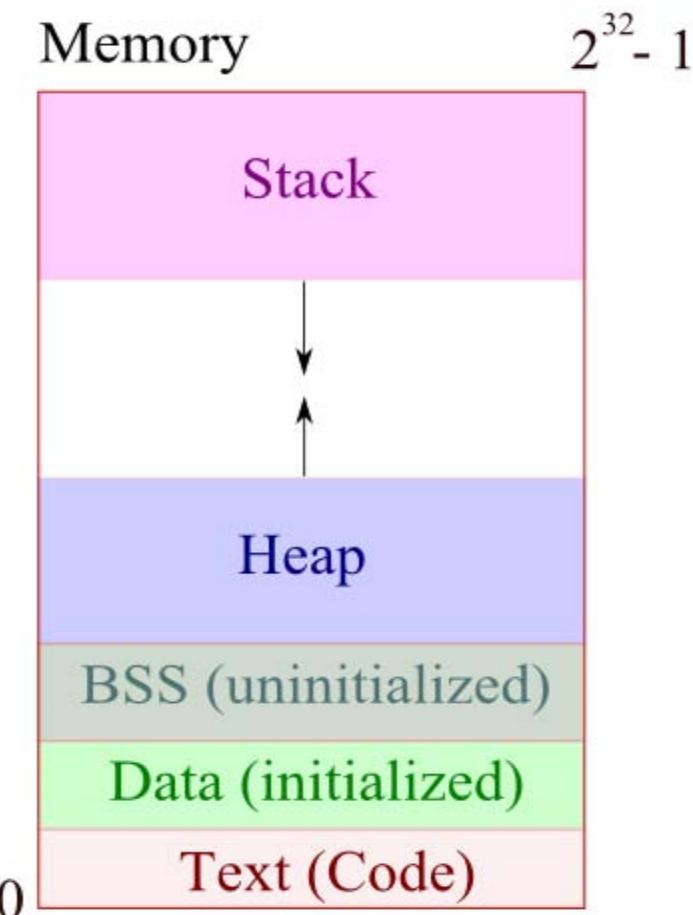
# Heap

- Heap is a chunk of memory that users can use to dynamically allocated memory.
- Lasts until freed, or program exits.

# Stack

- Stack contains local variables from functions and related book-keeping data. LIFO structure.
  - Function variables are pushed onto stack when called.
  - Functions variables are popped off stack when return.

# Memory Layout



Memory Layout diagram courtesy of [bogotobogo.com](http://bogotobogo.com), and used with permission.

# Call Stack

- Example: DrawSquare called from main()

```
void DrawSquare(int i){
 int start, end, //other local variables
 DrawLine(start, end);
}
```

```
void DrawLine(int start, int end){
```

```
 //local variables
```

```
 ...
```

```
}
```

# Call Stack

Lower address

- Example:

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
```

```
int end){
```

```
//local variables
```

```
...
```

```
}
```

Top of Stack

Higher address

# Call Stack

Lower address

- Example:

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
```

```
int end){
```

```
//local variables
```

```
...
```

```
}
```

Top of Stack

int i (DrawSquare arg)

Higher address

# Call Stack

Lower address

- Example:

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
int end){
```

```
 //local variables
```

```
 ...
```

```
}
```

Top of Stack

main() book-keeping

int i (DrawSquare arg)

Higher address

# Call Stack

Lower address

- Example:

```
void DrawSquare(int i){
```

int start, end, ...

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
int end){
```

//local variables

...

```
}
```

DrawSquare  
stack frame

Top of Stack

local variables (start, end)

main() book-keeping

int i (DrawSquare arg)

Higher address

# Call Stack

Lower address

- Example:

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
int end){
```

```
 //local variables
```

```
 ...
```

```
}
```

DrawSquare  
stack frame

Top of Stack

start, end (DrawLine args)

local variables (start, end)

main() book-keeping

int i (DrawSquare arg)

Higher address

# Call Stack

- Example:

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
int end){
```

```
//local variables
```

```
...
```

```
}
```

DrawSquare  
stack frame

Lower address

Top of Stack

DrawSquare book-keeping

start, end (DrawLine args)

local variables (start, end)

main() book-keeping

int i (DrawSquare arg)

Higher address

# Call Stack

- Example:

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
```

```
int end){
```

```
//local variables
```

```
...
```

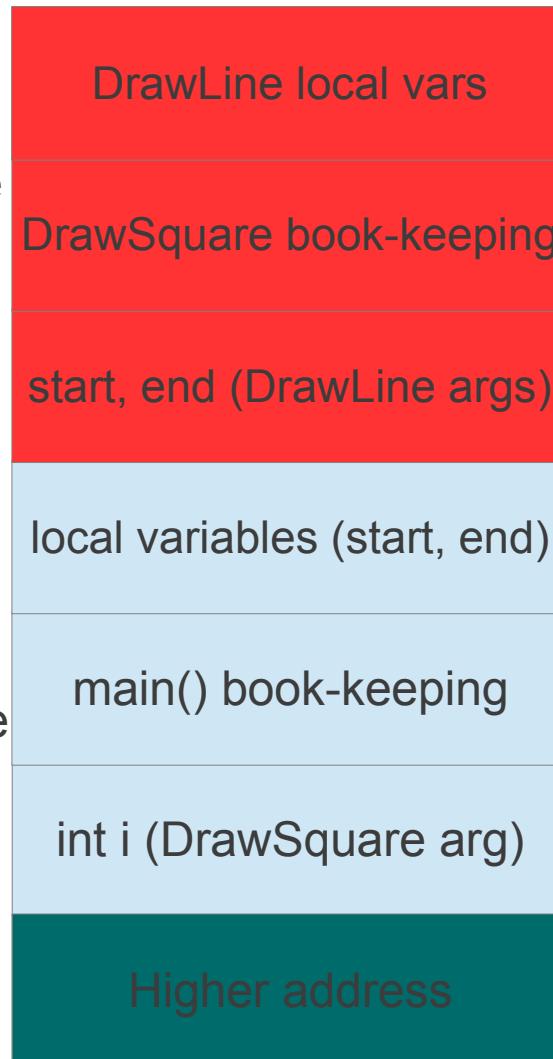
```
}
```

DrawLine  
stack  
frame

DrawSquare  
stack frame

Lower address

Top of Stack



# Call Stack

- Example: **DrawLine** returns

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
```

```
int end){
```

```
 //local variables
```

```
 ...
```

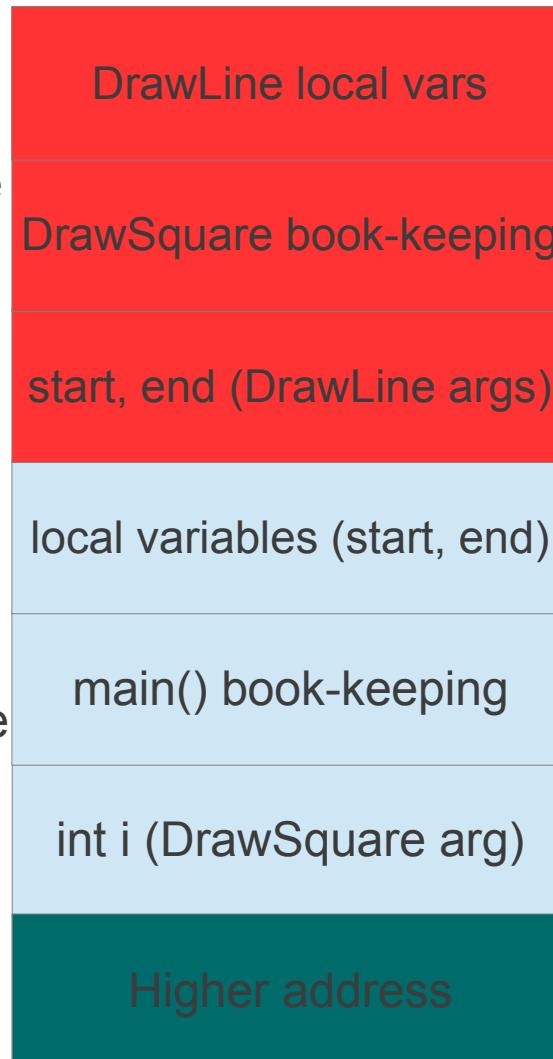
```
}
```

DrawLine  
stack  
frame

DrawSquare  
stack frame

Lower address

Top of Stack



# Call Stack

Lower address

- Example: **DrawLine** returns

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
```

```
int end){
```

```
 //local variables
```

```
 ...
```

```
}
```

DrawSquare  
stack frame

Top of Stack

local variables (start, end)

main() book-keeping

int i (DrawSquare arg)

Higher address

# Call Stack

Lower address

- Example: `DrawSquare` returns

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
int end){
```

```
 //local variables
```

```
 ...
```

```
}
```

DrawSquare  
stack frame

Top of Stack

local variables (start, end)

main() book-keeping

int i (DrawSquare arg)

Higher address

# Call Stack

Lower address

- Example: `DrawSquare` returns

```
void DrawSquare(int i){
```

```
 int start, end, ...
```

```
 DrawLine(start, end);
```

```
}
```

```
void DrawLine(int start,
int end){
```

```
 //local variables
```

```
 ...
```

```
}
```

Top of Stack

Higher address

# Today...

- Computer Memory
- **Pointers/Addresses**
- Arrays
- Memory Allocation

# Pointers and Addresses



Courtesy of xkcd at <http://xkcd.com/138/>, available under a CC by-nc license

# Addresses

- Each variable represents an address in memory and a value.
- Address:  $\&variable$  = address of variable

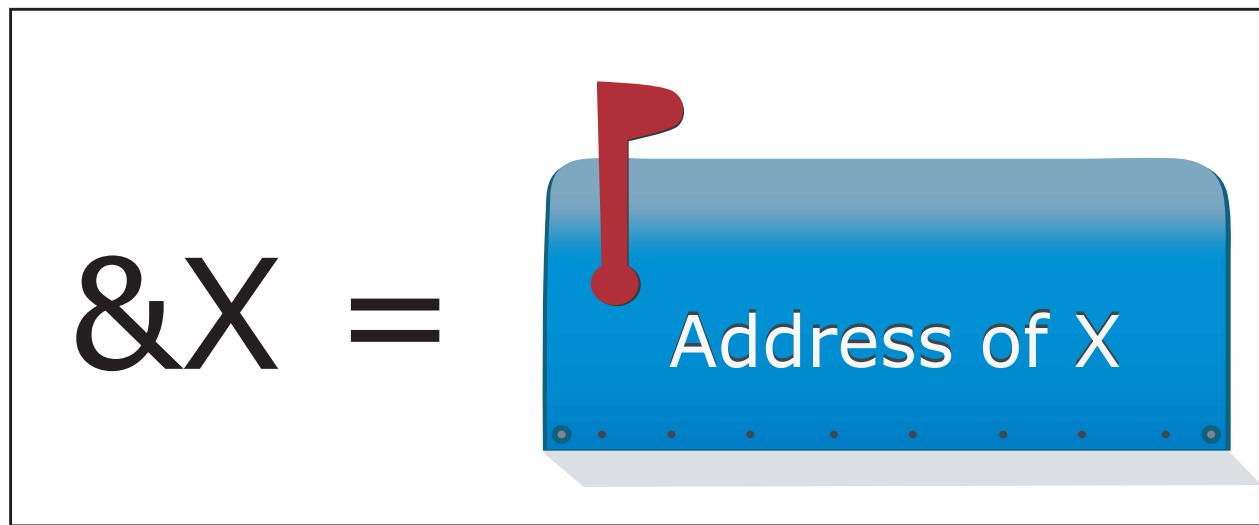


Image by MIT OpenCourseWare.

# Pointers

A pointer is a variable that “points” to the block of memory that a variable represent.

- Declaration: `data_type *pointer_name;`
- Example:

```
char x = 'a';
```

```
char *ptr = &x; // ptr points to a char x
```

# Pointers

A pointer is a variable that “points” to the block of memory that a variable represent.

- Declaration: `data_type *pointer_name;`
- Example:

```
char x = 'a';
```

```
char *ptr = &x; // ptr points to a char x
```

- Pointers are integer variables themselves, so can have pointer to pointers: `char **ptr;`

# Data type sizes

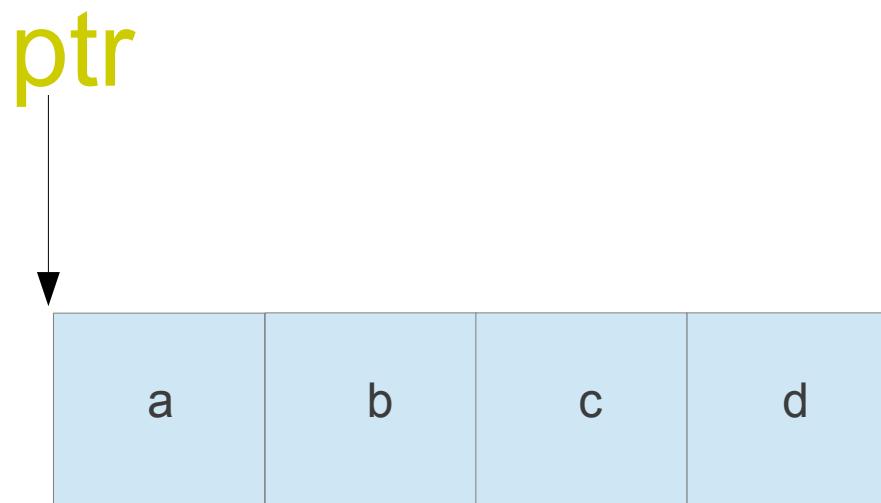
| Name              | Description                                                  | Size*  | Range*                                                         |
|-------------------|--------------------------------------------------------------|--------|----------------------------------------------------------------|
| char              | Character or small integer.                                  | 1byte  | signed: -128 to 127<br>unsigned: 0 to 255                      |
| short int (short) | Short Integer.                                               | 2bytes | signed: -32768 to 32767<br>unsigned: 0 to 65535                |
| int               | Integer.                                                     | 4bytes | signed: -2147483648 to 2147483647<br>unsigned: 0 to 4294967295 |
| long int (long)   | Long integer.                                                | 4bytes | signed: -2147483648 to 2147483647<br>unsigned: 0 to 4294967295 |
| bool              | Boolean value. It can take one of two values: true or false. | 1byte  | true or false                                                  |
| float             | Floating point number.                                       | 4bytes | +/- 3.4e +/- 38 (~7 digits)                                    |
| double            | Double precision floating point number.                      | 8bytes | +/- 1.7e +/- 308 (~15 digits)                                  |
| long double       | Long double precision floating point number.                 | 8bytes | +/- 1.7e +/- 308 (~15 digits)                                  |

# Dereferencing = Using Addresses

- Also uses \* symbol with a pointer. Confusing? I know!!!
- Given pointer ptr, to get value at that address, do: \*ptr
  - int x = 5;
- int \*ptr = &x;  
\*ptr = 6; // Access x via ptr, and changes it to 6  
printf("%d", x); // Will print 6 now
- Can use **void** pointers, just cannot dereference without casting

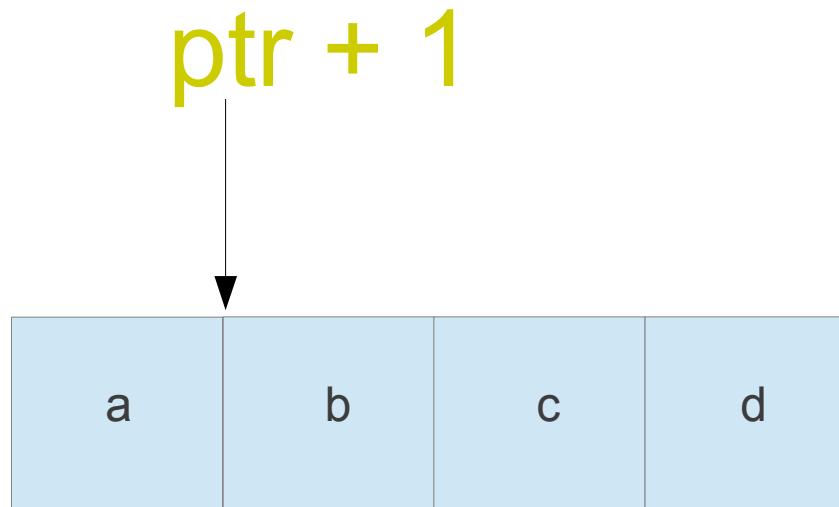
# Pointer Arithmetic

- Can do math on pointers
  - Ex: `char* ptr`



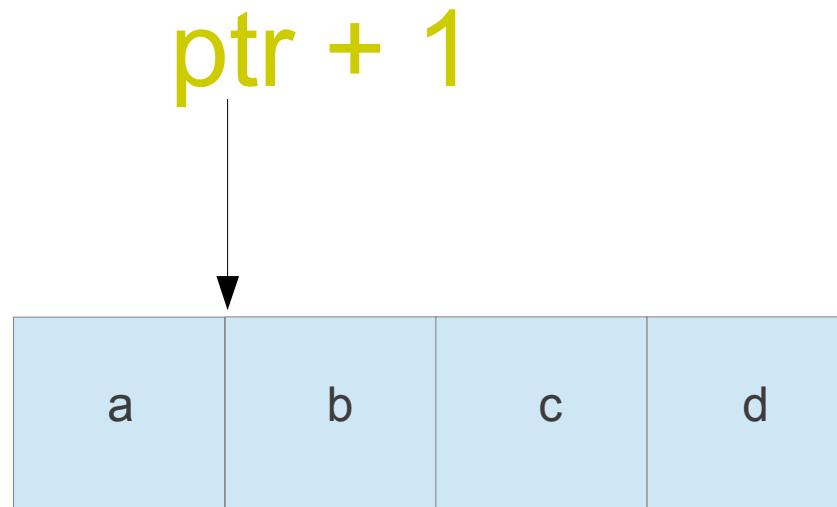
# Pointer Arithmetic

- Can do math on pointers
  - Ex:  $\text{char}^*$  ptr



# Pointer Arithmetic

- Can do math on pointers
  - Ex: `char* ptr`



`ptr+i` has value: `ptr + i * sizeof(data_type of ptr)`

# Pointer Arithmetic

- Can do math on pointers
  - $p1 = p2$ : sets  $p1$  to the same address as  $p2$
  - Addition/subtraction:
    - $p1 + c$ ,  $p1 - c$
  - Increment/decrement:
    - $p1++$ ,  $p1--$

# Why use pointers? They so confuzin...

- Pass-by-reference rather than value.

```
void sample_func(char* str_input);
```

- Manipulate memory effectively.

- Useful for arrays (next topic).

# Today...

- Computer Memory
- Pointers/Addresses
- **Arrays**
- Memory Allocation

# C Arrays (Statically Allocated)

- Arrays are really chunks of memory!
- Declaration:
  - `Data_type array_name[num_elements];`
- Declare array size, cannot change.

# C Arrays (Statically Allocated)

- Can initialize like:

- `int data[] = {0, 1, 2}; //Compiler figures out size`
- `int data[3] = {0, 1, 2};`
- `int data[3] = {1}; // data[0] = 1, rest are set to 0`
- `int data[3]; //Here, values in data are still junk`

`data[0] = 0;`

`data[1] = 1;`

`data[2] = 2;`

# Array and Pointers

- Array variables are pointers to the array start!

- `char *ptr;`

```
char str[10];
```

```
ptr = str; //ptr now points to array start
```

```
ptr = &str[0]; //Same as above line
```

- Array indexing is same as dereferencing after pointer addition.

- `str[1] = 'a'` is same as `*(str+1) = 'a'`

# C-Style Strings

- No string data type in C. Instead, a string is interpreted as a null-terminated char array.
- Null-terminated = last char is null char '\0', not explicitly written

```
char str[] = "abc";
```



- String literals use “ ”. Compiler converts literals to char array.

# C-Style Strings

- Char array can be larger than contained string

```
char str[5] = "abc";
```

|   |   |   |    |  |
|---|---|---|----|--|
| a | b | c | \0 |  |
|---|---|---|----|--|

- Special chars start with '\':
  - \n, \t, \b, \r: newline, tab, backspace, carriage return
  - \\", \', \": backslash, apostrophe, quotation mark

# String functionalities

- `#include <string.h>`
- char pointer arguments: `char str1[14]`
  - `char* strcpy(char* dest, const char* source);`  
`strcpy(str1, "hakuna ");`
  - `char* strcat(char* dest, const char* source);`  
`strcat(str1, "matata"); //str1 now has "hakuna matata"`
- More in documentation...

# Today...

- Computer Memory
- Pointers/Addresses
- Arrays and Strings
- **Memory Allocation**

# Dynamic Allocation

- `#include <stdlib.h>`
- **sizeof** (a C language keyword) returns number of bytes of a data type.
- **malloc/realloc** finds a specified amount of free memory and returns a void pointer to it.
  - `char * str = (char *) malloc( 3 * sizeof(char) );`  
`strcpy(str, "hi");`
  - `str = (char *) realloc( str , 6 * sizeof(char) );`  
`strcpy(str, "hello");`

# Dynamic Deallocation

- `#include <stdlib.h>`
- **free** declares the memory pointed to by a pointer variable as free for future use:

```
char * str = (char *) malloc(3 * sizeof(char));
strcpy(str, "hi");
... use str ...
free(str);
```

# Dynamically Allocated Arrays

- Allows you to avoid declaring array size at declaration.
  - Use malloc to allocate memory for array when needed:
    - `int *dynamic_array;`
- ```
dynamic_array = malloc( sizeof( int ) * 10 );  
dynamic_array[0]=1; // now points to an array
```

Summary

- Memory has stack and heap.
- Pointers and addresses access memory.
- Arrays are really chunks of memory. Strings are null-terminated char arrays.
- C allows user memory allocation. Use malloc, realloc and free.

MIT OpenCourseWare
<http://ocw.mit.edu>

6.S096 Introduction to C and C++

IAP 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.



MIT OpenCourseWare

6.S096 | January IAP 2013 | Undergraduate

Introduction To C And C++

≡ Menu

More Info

Feedback

Lectures and Assignments

C Memory Management

Topics: Computer memory layout (heap, stack, call stack), pointers and addresses, arrays, strings, and manual memory allocation/deallocation.

Lecture Notes

[Lecture 3: C Memory Management \(PDF\)](#)

Lab Exercises

The primary goal of this lab period is to introduce pointers, addresses, arrays, and manual memory management.

Exercise 1

```
#include <stdio.h>

void square(int num) {
    num = num * num;
}

int main() {
    int x = 4;
    square(x);
    printf("%d\n", x);
    return 0;
}
```

Take a look at the above code. In lecture, I pointed out that function variables are passed by value, not reference. So this program will currently print out 4. Compile the code to confirm this.

Use pointers and addresses to modify the code so x is passed by reference instead and is squared. This will involve changes to the square function that *does not* involve changing void to

int and giving square a return statement. Make sure your code compiles with `-Wall` flag without warnings.

Exercise 2

While coding up this exercise, listening to Hakuna Matata, I was so worry-free I forgot how to use C!

Fix the following code so that it creates a string `str` and copies “`hakuna matata!`” into it.

```
#include <stdio.h>

void main() {
    char str[];
    ???(, "hakuna matata!"); // this line should copy "hakuna matata!"
                                // into our char array
    printf("%s\n", str);
    // Anything else?
}
```

After confirming your fix works, change the code to use heap memory instead of the stack. Remember, everything you `malloc` you must also `free`!

Assignment 3

Problem 1

sort (C)

In `sort.c`, I've implemented a basic implementation of insertion sort (not too efficient, but a very simple sorting algorithm). Look at and understand the code (read comments), and put the proper argument data type for the `sort` function's first argument. Compile and run the code to make sure it works (it sorts the numbers).

Now, replace all array index access (places where I access the array with `[]`, such as in `A[i]`) in the entire program by using pointer addition instead. Also, where I create the array (line 34, `int array[n];`), replace that with an array creation using `malloc` (of same size).

Hint: Since we're using `malloc`, we must also make another change!

Make sure your program compiles and runs correctly (numbers are sorted).

Problem 2

resize (C)

The purpose of `resize.c` is to create an initial array of a user-specified size, then dynamically resize the array to a new user-specified size. I've given a shell of the code, including code to get user-specified sizes as `ints`.

However, the code is missing a few things. You must manage the memory for the array! Look at the comments in the code that describe what should be done and fill in blanks. Make sure the program compiles and runs as expected.

Solutions

[Assignment 3 solution \(PDF\)](#)



Over 2,500 courses & materials

Freely sharing knowledge with learners and educators around the world. [Learn more](#)

[Accessibility](#)

[Creative Commons License](#)

[Terms and Conditions](#)

Proud member of: The logo for Open Education GLOBAL, featuring a stylized globe icon followed by the text "Open Education GLOBAL".



© 2001–2025 Massachusetts Institute of Technology

6.S096: Introduction to C/C++

Frank Li, Tom Lieber, Kyle Murray

Lecture 4: Data Structures and Debugging!

January 17, 2012

Today...

- Memory Leaks and Valgrind Tool
- Structs and Unions
- Opaque Types
- Enum and Typedef
- GDB Debugging Tool

Today...

- **Memory Leaks and Valgrind Tool**
- Structs and Unions
- Opaque Types
- Enum and Typedef
- GDB Debugging Tool

Memory Issues (Live Demos!)

- Illegal memory accesses (segmentation faults)
- Stack overflow
 - Infinite recursions
- Double frees
- Dangerous use of uninitialized variables (see Lec 2)
- Memory leaks

Memory Leaks

- Since memory is user managed, there can be mistakes.
- Memory leaks = allocated memory that's not freed.
- Why it matters?

Valgrind

- A memory profiling tool. Helps you find memory leaks.
- Compile with debug mode (using -g flag).

```
gcc -g hello.c -o hello
```

- Includes tools such as memcheck, cachegrind, callgrind, etc...
- Command line:
 - valgrind --tool=*tool_name* ./program_name
 - Example: valgrind --tool=memcheck ./hello_world

Valgrind No Leaks Demo

- \$ valgrind --tool=memcheck ./hello_world
...
=18515== malloc/free: in use at exit: 0 bytes in 0 blocks.
==18515== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.
==18515== For a detailed leak analysis, rerun with: --leak-check=yes

Valgrind Leak Demo

- #include <stdlib.h>

```
int main()
```

```
{
```

```
    char *x = (char*) malloc(100); //Mem Leak!
```

```
    return 0;
```

```
}
```

Valgrind No Leaks Demo

- \$ valgrind --tool=memcheck ./mem_leak

...

```
==12196== HEAP SUMMARY:
```

```
==12196==    in use at exit: 100 bytes in 1 blocks
```

```
==12196==    total heap usage: 1 allocs, 0 frees, 100 bytes allocated
```

```
==12196==
```

```
==12196== LEAK SUMMARY:
```

```
==12196==    definitely lost: 100 bytes in 1 blocks
```

```
==12196==    indirectly lost: 0 bytes in 0 blocks
```

```
==12196==    possibly lost: 0 bytes in 0 blocks
```

```
==12196==    still reachable: 0 bytes in 0 blocks
```

```
==12196==    suppressed: 0 bytes in 0 blocks
```

More Valgrind Functionalities

- Can also detect invalid pointer use
 - `char *arr = malloc(10);`
`arr[10] = 'a'`
- Using uninitialized variables
 - `int x;`
`if(x==0){...}`
- Double/invalid frees
- Valgrind doesn't check bounds on statically allocated arrays though!

Today...

- Memory Leaks and Valgrind Tool
- **Structs and Unions**
- Opaque Types
- Enum and Typedef
- GDB Debugging Tool

User Defined Data Types

- C has no objects, but has data structures that can help fill in roles. Only contains data.
- Structures and unions are like data objects
 - Contains groups of basic data types
 - Can be used like any normal type
 - Not true objects b/c they can't contain functions

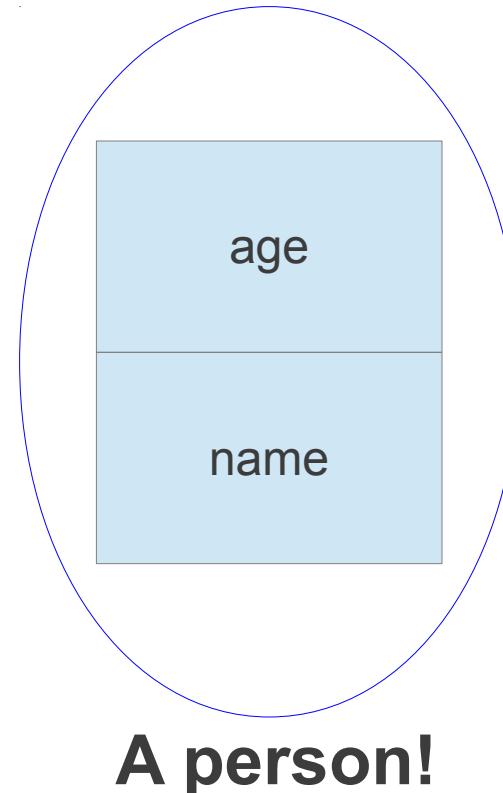
Structures

- Declaring structs

```
struct person{  
    int age;  
    char name[50];  
};
```

```
struct person joe;
```

- Accessing members:
 - operator
 - `instance_name.struct_member`
 - `joe.age`



Structures

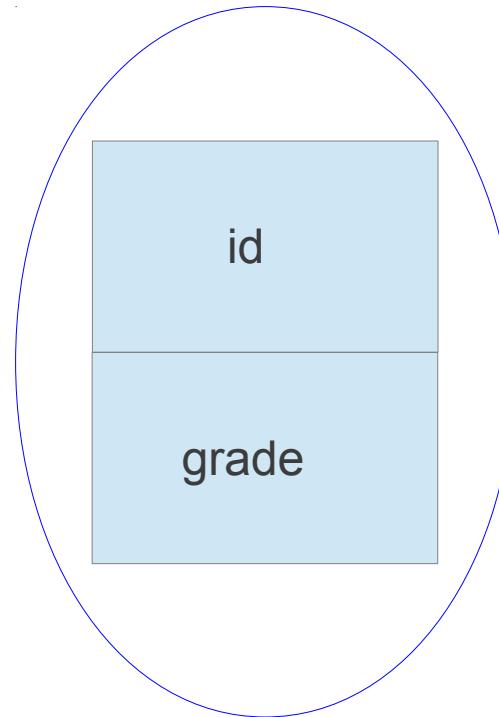
- Initializing structs

```
▫ struct student {  
    int id;  
    char grade;  
};
```

```
struct student frank= {1, 'A'};
```

Or

```
struct student frank;  
frank.id = 1;  
frank.grade = 'A';
```



**Hmm...is that all I am?
A number and a letter?**

:(
:(

Structures

- Struct pointers:
 - `struct student * frank= malloc(sizeof(struct student));`
- Accessing members of struct pointers: ->
 - `frank->grade = 'A' // Hooray (for me)!`
- Can have structs within structs. Just use more . or -> operators to access inner structs.

Struct Memory

- Struct size != sum of member sizes
- All members must “align” with largest member size
- Each member has own alignment requirements
 - Ex: char = 1-byte aligned.
 - short = 2-byte aligned.
 - int = 4-byte aligned. ←Refer to documentation

Struct Memory

- Blackboard Example:

```
struct x{  
    char a; //1-byte, must be 1-byte aligned  
    short b; //2-bytes, must be 2-byte aligned  
    int c; // Biggest member (4-bytes). X must be 4-byte  
           // aligned  
    char d;  
}
```

Unions

- Can only access one member at a time. Union stores all data in same chunk of memory.

```
union data{
```

```
    int x; char y;
```

```
};
```

```
union data mydata;
```

```
mydata.y = 'a';
```

```
mydata.x = 1;
```

```
printf("%d\n", mydata.x) // Will print out 1
```

```
printf("%c\n", mydata,y) // Will print out JUNK!
```

Common Use of Unions

- Union within struct

```
struct student{  
    union grade{  
        int percentage_grade; //ex: 99%  
        char letter_grade; // 'B'  
    };  
    int grade_format; // I set to 0 if grade is int, 1 if grade is  
    char  
};  
struct student frank;  
frank.grade.percentage_grade = 90;  
frank.grade_format = 0;
```

Today...

- Memory Leaks and Valgrind Tool
- Structs and Unions
- **Opaque Types**
- Enum and Typedef
- GDB Debugging Tool

Opaque Types

- Type exposed via pointers where definition can still change. Ex: Can change struct person definition in test.c without recompiling my_file.c

test.h:

```
struct person;
```

test.c:

```
struct person{  
    ... //def here  
};
```

my_file.c:

```
#include "test.h"  
int person_function(struct person * ptr){  
    ...  
}
```

Today...

- Memory Leaks and Valgrind Tool
- Structs and Unions
- Opaque Types
- **Enum and Typedef**
- GDB Debugging Tool

Typedef

- Typedef assigns alternate name to existing type.

- `typedef existing_type alternate_name`

```
typedef int seconds_t;
```

```
seconds_t x = 3;
```

```
typedef struct person person;
```

```
person frank; // instead of struct person frank
```

- Good for shorthand and code readability

- Opaque types

```
typedef struct person* person;
```

```
int func(person me){ ...}
```

Enum

- Define own variable type with a set of int values

```
enum time_t { morning, noon, afternoon, night};
```

```
enum time_t class = morning;
```

```
if (class == afternoon) { ... }
```

- What actually happens is enum values are mapped to increasing sequence of int:

- morning = 0, noon = 1, afternoon = 2, night = 3

- You can explicitly assign int values:

```
enum time_t { morning, noon=2, afternoon, night};
```

```
morning = 0, noon = 2, afternoon = 3, night = 4
```

Why use Enum

- Like a #define variable but is actually a C element (has a type, obeys scoping rules)

```
#define NORTH 0
#define EAST 1
#define SOUTH 2
#define WEST 3

int direction = SOUTH;
//Compiler sees:
//int direction = 2
```

```
enum dir_t{
    NORTH,
    EAST,
    SOUTH,
    WEST
};

typedef enum dir_t dir_t;
dir_t direction = SOUTH;
```

Today...

- Memory Leaks and Valgrind Tool
- Structs and Unions
- Opaque Types
- Enum and Typedef
- **GDB Debugging Tool**

Debugger (Your LifeSaver)

- Compile with –g flag (debug mode)

```
gcc -g hello.c -o hello
```

- Command-line debugger: gdb ./prog_name

```
gdb ./hello
```

GDB Example

```
C:\Documents and Settings\U0030494\Desktop>gcc -g helloworld.c -o helloworld.exe

C:\Documents and Settings\U0030494\Desktop>helloworld
Hello World!
C:\Documents and Settings\U0030494\Desktop>gdb helloworld
GNU gdb (GDB) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from C:\Documents and Settings\U0030494\Desktop/helloworld.exe...
done.
(gdb) r
Starting program: C:\Documents and Settings\U0030494\Desktop/helloworld.exe
[New Thread 5848.0x12a0]
Hello World!
Program exited with code 014.
(gdb) q

C:\Documents and Settings\U0030494\Desktop>
```

Debugger (Live Demo!)

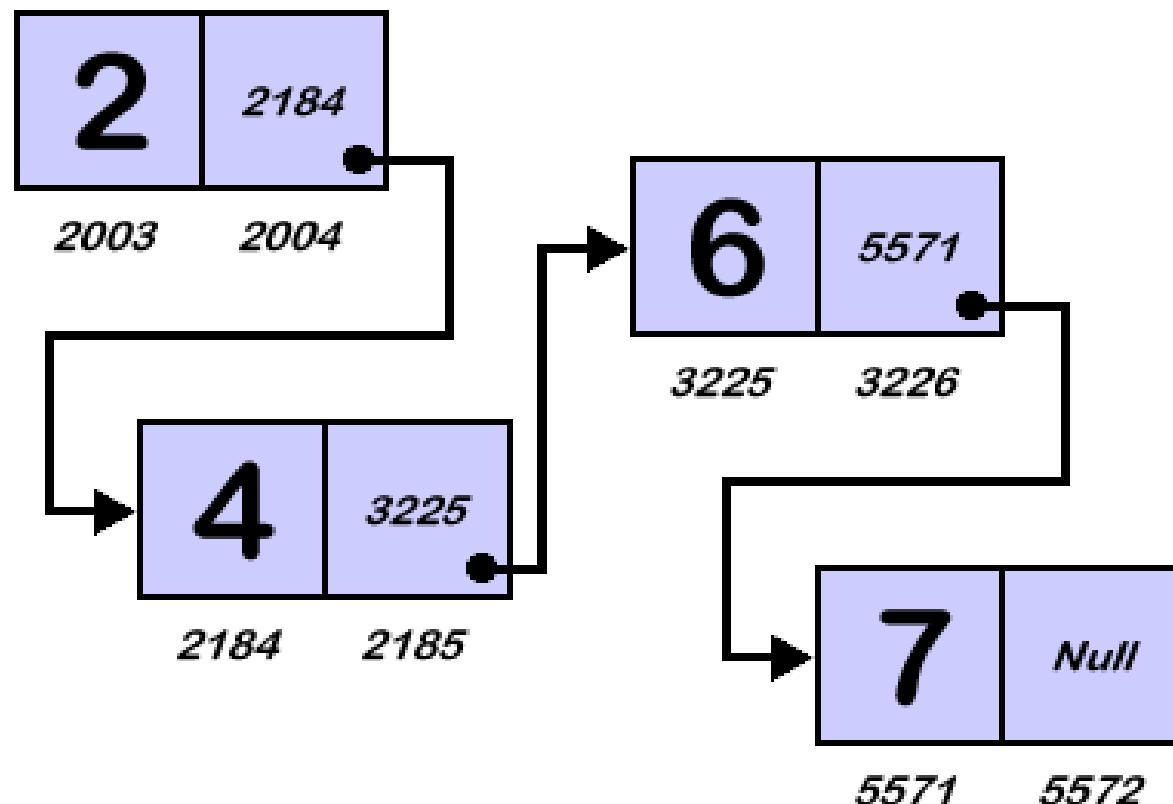
Useful commands:

- q/quit: exit GDB
- r/run: run program
- b/break (**filename:**)**linenumber**: create a breakpoint at the specified line.
- s/step: execute next line, or step into a function
- c/continue: continue execution
- p **variable**: print the current value of the variable
- bt: print the stack trace (very useful!)

Summary

- Use Valgrind and GDB to find errors/memory bugs
- C structs and unions are like data “objects”
- Opaque types allow flexibility in struct/union usage
- Enum and Typedef saves you time!

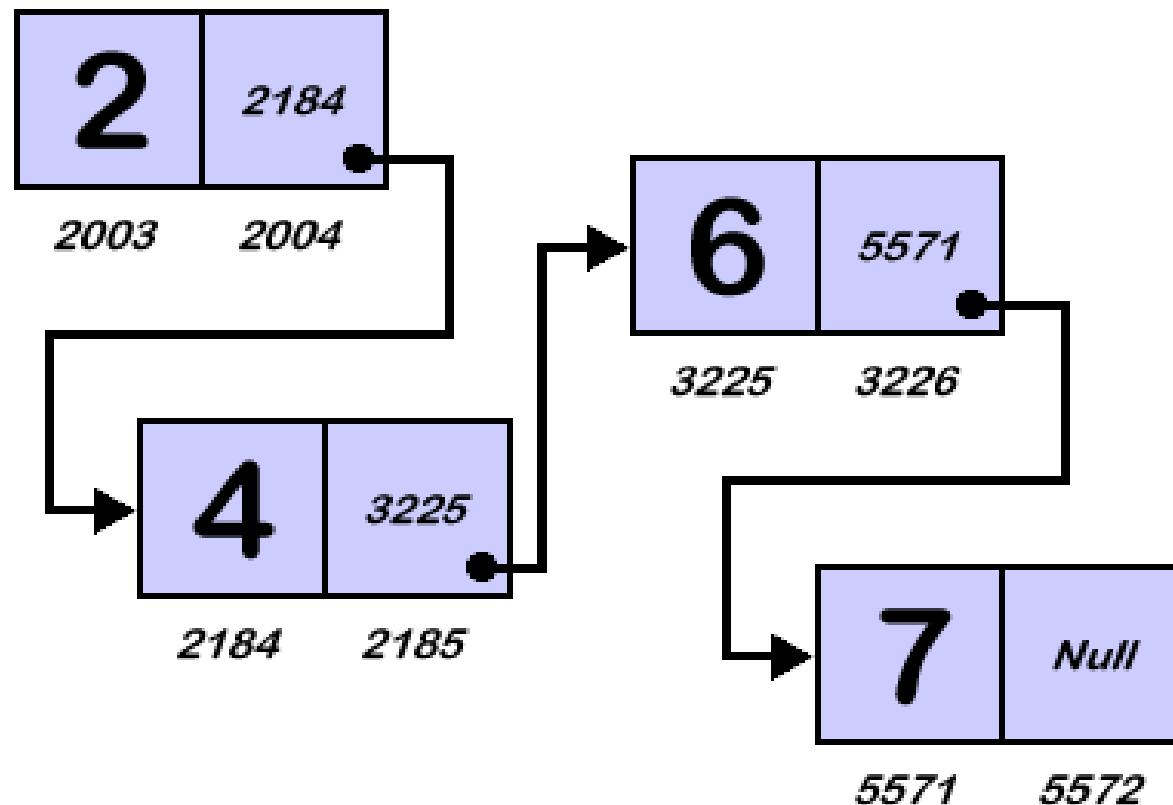
Advanced Data Type Example



Courtesy of Osman Balci, Virginia Tech Center for Innovation in Learning. Used with permission.

Image from: <http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/Lesson.html>

The Linked List!

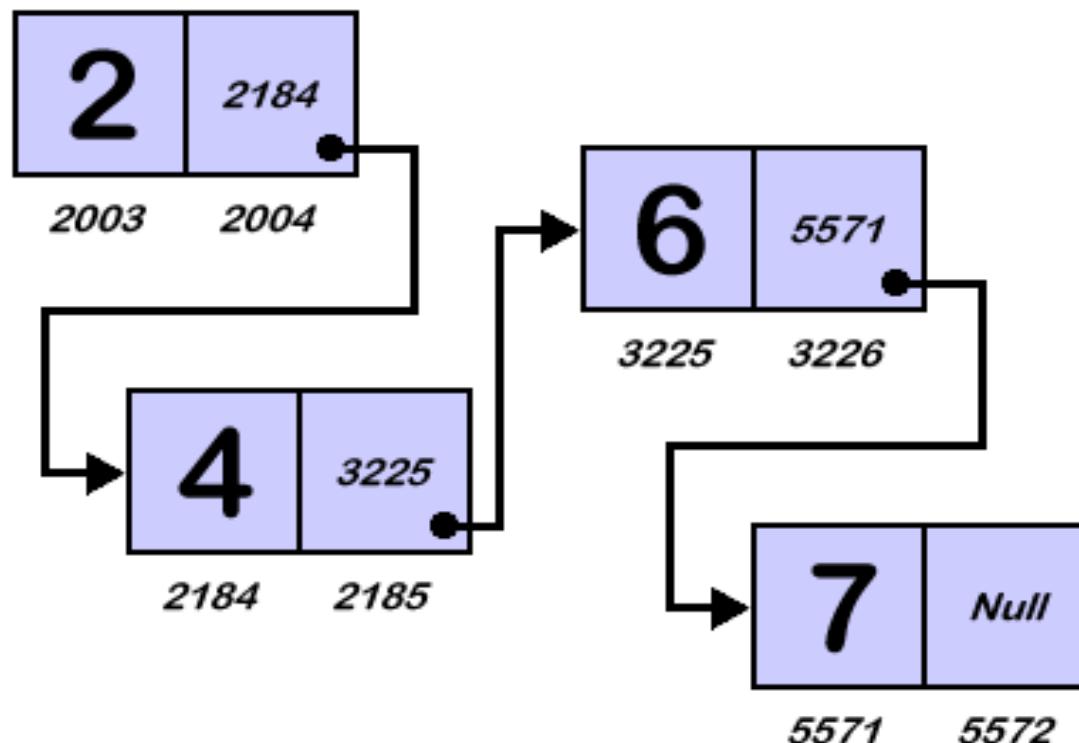


Courtesy of Osman Balci, Virginia Tech Center for Innovation in Learning. Used with permission.

Image from: <http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/Lesson.html>

Singly Linked Lists

- Singly linked list is a sequential list of nodes where each node contains a value and a link/pointer to the next node.



Courtesy of Osman Balci, Virginia Tech Center for Innovation in Learning. Used with permission.

Image from: <http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/Lesson.html>

Singly Linked Lists

- Nodes:

```
struct node{  
    int data;  
    struct node* next;  
};  
  
typedef struct node node;  
  
node* head = NULL; //Points to beginning of list. Set  
                  // to null initially.
```

Singly Linked Lists

- Adding new data to list

```
node* add_data(int data){  
    node* new_node = (node*) malloc(sizeof(node));  
    new_node->data = data;  
    new_node->next = head;  
    head = new_node;  
    return new_node;  
}
```

Singly Linked Lists

- Searching through list

```
node * find_data(int data){  
    node* current;  
  
    for( current = head; current->next!=NULL;  
         current= current->next){  
  
        if(current->data == data) return current;  
    }  
  
    return NULL;  
}
```

Singly Linked Lists

- Removing a certain data value

```
void rm_data(int data){  
    //Special case if the head has the data  
    if( head->data == data ) {  
        node* tmp = head;  
        head = head->next;  
        free(tmp);  
        return;  
    }  
    ...//Code continues on next slide  
}
```

Singly Linked Lists

- Removing a certain data value

```
void rm_data(int data){  
    ...//Code from previous slide  
    node* prev, *current;  
    for(current = head; current->next!=NULL; current= current->next){  
        if(current->data == data){  
            prev->next = current->next;  
            free(current);  
            return ;  
        }  
        prev = current;  
    }  
}
```

MIT OpenCourseWare
<http://ocw.mit.edu>

6.S096 Introduction to C and C++

IAP 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.



OpenCourseWare

6.S096 | January IAP 2013 | Undergraduate

Introduction To C And C++

[Menu](#) [More Info](#) [Feedback](#)

Lectures and Assignments

Data Structures, Debugging

Topics: Using structs, unions, typedef, and enums, and how to debug with Valgrind and GDB.

Lecture Notes

[Lecture 4: Data Structures, Debugging \(PDF\)](#)

Lab Exercises

The primary goal of this lab period is to introduce debugging tools, and use of unions/structs.

Exercise 1

Download and install [Valgrind](#) on your system, if it's not already. To test if you have Valgrind, run `valgrind --version`. It should print the version of Valgrind that is installed.

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

void fn()
{
    int* x = malloc(10 * sizeof(int));
    printf("%d", *x);
    x[10] = 0;
}

int main()
{
    fn();
    return 0;
}
```

There are 3 sources of memory errors in this code. Run valgrind to determine what they are (although I suspect you can probably tell from the code anyways).

Exercise 2

Use a union to print the individual bytes of an `int`. (Hint: Recall the size of `int`s and other data types.)

Exercise 3

Determine how much memory is required for each of the `struct`s below. How much of that memory is padding between the members?

```
struct X
{
    short s;
    int i;
    char c;
};

struct Y
{
    int i;
    char c;
    short s;
};

struct Z
{
    int i;
    short s;
    char c;
};
```

Assignment 4

Today's assignment combines the material from the past few lectures. Your job is to fill in the skeleton code we provide. I have commented the code with what each section should do.

[Assignment 4 files \(ZIP\)](#) (This ZIP file contains: 2 .c files and 1 .h file.)

You can learn more about binary search trees and find pseudo-code on [the binary search tree page on Wikipedia](#).

Your job is to implement a binary search tree, a data structure of connected nodes with a tree shape. Each node has a node identifier (a number), data (payload), and 2 children (left and right). The children are other nodes referenced with a pointer, with the constraint that the left node's ID is less than the parent node's ID, and the right node's ID is larger than the parent node ID. No two nodes will have the same identifier. A node can have less than two children; in that case, one or more of its child pointers can be `NULL`.

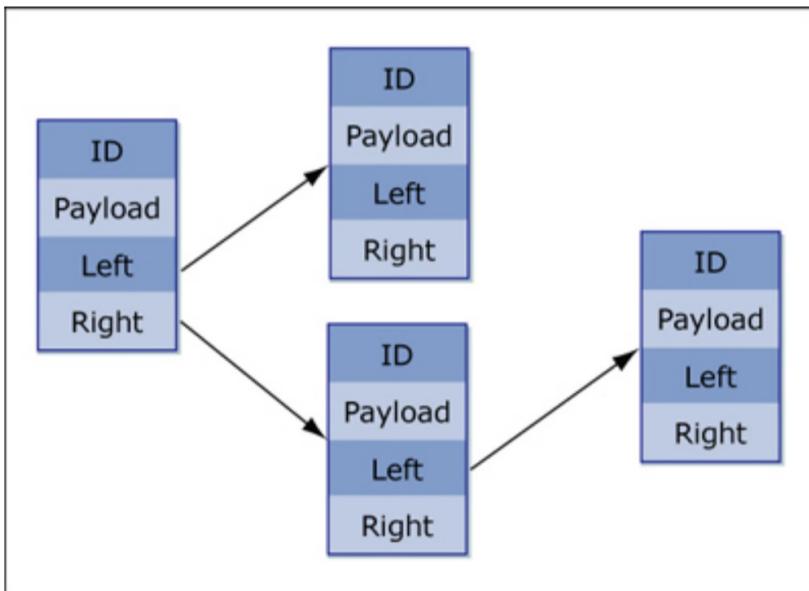


Image by MIT OpenCourseWare.

`user.c` contains the `main()` function. We will replace this function with one for grading. You should use your `main()` function to test that your functions to insert into and search the binary tree work correctly.

This is a C/C++ course, not an algorithms course, but if you want a challenge, try implementing node deletion as well!

Your job is to complete the data structure and function declarations in `bintree.h`, then complete the implementation of your functions in `bintree.c`. If you want to define additional functions to simplify your program, that's fine. You cannot change the return types or argument types of the included functions, though. Even though we don't require the deletion function, make sure to free all memory you allocate!

Make sure your program compiles without warning, runs, and *definitely* use `valgrind` to ensure you have no memory leaks.

```
$ gcc -Wall -std=c99 user.c bintree.c -o bintree
$ ./bintree
<your test output>
```

Solutions

Solutions are not available for this assignment.



Over 2,500 courses & materials

Freely sharing knowledge with learners and educators around the world. [Learn more](#)

[Accessibility](#)

[Creative Commons License](#)

[Terms and Conditions](#)

Proud member of:  The logo for Open Education GLOBAL, featuring a blue stylized 'G' icon followed by the text 'Open Education' and 'GLOBAL' stacked vertically.



© 2001–2025 Massachusetts Institute of Technology

6.s096

Lecture 5

Today

- C++ (!)
 - Compiling
 - Memory management
- Classes
- Templates

C++

C++

- Bjarne Stroustrup
- 1983
- Object-oriented (but later than Simula, Smalltalk)
- Like C, but introduces real objects and classes
- (plus loads of other features (kitchen sink?))

g++ (C++ compiler)

- Very similar to gcc.

```
g++ -o test test.cpp  
./test  
=> hi from C++
```

```
#include <stdio.h>  
  
int main(){  
    printf("hi from C++\n");  
}
```

Wait... was that really C++?

- Yes.
- C++ is pretty close to being a superset of C.
- We know C, thus we'll build from that knowledge to learn C++.

new memory management syntax

- The new operator allocates space on the heap.
- new and delete take the place of malloc and free.

```
int * numArray = new int[100];  
delete numArray;
```

```
struct foo * bar = new struct foo; // delete later
```

Classes

Why classes?

- Modularity
- Objects (data + behavior)
- Lets programmers (you) define behavior for your own data

Basic Class Example

```
#include <stdio.h>

class Rectangle {
    int * width;
    int * height;

public:
    Rectangle(int, int); // constructor
    ~Rectangle(); // destructor
    void printMe(){ // 'method' / member function
        printf("Dimensions: %d by %d.\n", *width, *height);
    }
};
```

```
Rectangle::Rectangle(int w, int h){
    // constructor definition
    width = new int;
    height = new int;
    *width = w;
    *height = h;
}

int main(){
    Rectangle box(5, 7);
    box.printMe();
}
```

Constructors and Destructors

- This destructor should have fit on the last slide...:
- Since we explicitly allocated something with new, we must also explicitly de-allocate it.
- Rectangle itself is automatically deallocated when it goes out of scope.

```
Rectangle::~Rectangle(){
    delete width;
    delete height;
}
```

Default constructors

```
Rectangle::Rectangle(){      // no arguments needed!
    width = new int;
    height = new int;
    *width = 5;
    *height = 5;
}
```

Templates

- Syntax for making code more flexible.
- Similar in spirit to Java's generics.
- Applied at compile-time, like C macros (the preprocessor).
- Can be applied to classes, functions.
- Trivia: language of templates is Turing complete.

Function Template Example

```
template <class typeParam>
typeParam max(typeParam a, typeParam b){
    return (a > b ? a : b);
}

int main(){
    int a = 3, b = 7;
    double c = 5.5, d = 1.5;
    printf("%d\n", max(a, b)); // 7
    printf("%f\n", max(c, d)); // 5.5
}
```

Class Template Example

```
template <class T>
class mypair {
    T a, b;
public:
    mypair(T first, T second){
        a = first;
        b = second;
    }
    T getmax();
};
```

```
template <class T>
T mypair<T>::getmax(){
    return a > b ? a : b;
}

int main(){
    mypair<int> myints(100, 75);
    printf("%d\n",
    myints.getmax()); // 100
```

MIT OpenCourseWare
<http://ocw.mit.edu>

6.S096 Introduction to C and C++
IAP 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.



MIT OpenCourseWare

6.S096 | January IAP 2013 | Undergraduate

Introduction To C And C++

≡ Menu

More Info

Feedback

Lectures and Assignments

C++ Introduction, Classes, and Templates

Lecture Notes

[Lecture 5: C++ Introduction, Classes, and Templates \(PDF\)](#)

Lab Exercises

There were no lab exercises to accompany this lecture.

Assignment 5

Problem 1

Now that we've transitioned from learning C to learning C++, we should be able to transition some C-style code that uses `struct`, `typedef`, and ordinary functions into C++ code that uses a single class to do the same job.

Download the C code below, and create a new file, `p1_grades.cpp`.

[p1_grades \(C\)](#)

Your job is to create a C++ class named `Grade` that has the same functionality as the old `struct Grade` and associated functions in the original C file. When you are finished, your C++ file should only have one function definition outside of the class: `main()`. You should use the following definition of main:

```
int main() {
    Grade g;
    int percent;

    printf("Enter two grades separated by a space. Use a percentage for
the first and letter for the second: ");
    scanf("%d", &percent);
    scanf("\n");

    g.setByPercent(percent);
    g.print();

    g.setByLetter(getchar());
    g.print();

    return 0;
}
```

The names and interface of your Grade class should match the way the Grade instance is being used in the main function above. (That is, it should have member functions named `setByPercent`, etc. that are compatible with the use of those functions in `main.`)

Note the general way the grade program works: The user runs the program and is asked to enter two pieces of input. The first is an integer between 1 to 100 representing a percentage grade. The second input, separated on the command line by a space, is a letter grade (A, B, C, D, F). The output is two lines; each line shows the original and converted forms of the grade. So, for example, entering the input “**100 F**” would generate the lines:

```
Grade: 100: A
Grade: 45: F
```

The two input grades aren’t related (a 100 isn’t an F!). Instead, the inputs are used to show both directions of the conversion. Letter grades are converted to “nearby” percentages that fall into the right range. Don’t worry about changing the grading logic. You can use the existing scale / system, including reusing `GRADE_MAP`.

Make sure your program compiles without warning, runs, and *definitely* use `valgrind` to ensure you have no memory leaks.

```
$ g++ -Wall p1_grades.cpp -o p1_grades
$ ./p1_grades
< provide a [percentageGrade] [letterGrade] input pair (like "97 D") >
<your test output>
```

Problem 2

In this problem, you will be converting a class that is specialized for integers into a templated class that can handle many types, including integers and structs. You will create a templated class named `List` that correctly initializes, manages, and de-allocates an array of a specified length. This is a nice class because the normal C arrays we've seen do not keep track of their length at runtime, but the class we're building will do that for us!

p2_templates (CPP)

When you're finished writing your templated `List` class, you should change your `main()` function to this code below (this is the same code that's in the starter file, `p2_templates.cpp`):

```
int main() {
    List<int> integers(10);
    for (int i = 0; i < integers.length; i++) {
        integers.set(i, i * 100);
        printf("%d ", integers.get(i));
    }
    printf("\n"); // this loop should print: 0 100 200 300 400 500 600 700 800 900

    List<Point *> points(5);
    for (int i = 0; i < points.length; i++) {
        Point *p = new Point;
        p->x = i * 10;
        p->y = i * 100;
        points.set(i, p);
        printf("(%d, %d) ", points.get(i)->x, points.get(i)->y);
        delete p;
    }
    printf("\n"); // this loop should print: (0, 0) (10, 100) (20, 200) (30, 300)
}
```

This main function makes use of a `typedef struct` called `Point`. Here's its definition:

```
typedef struct Point_ {
    int x;
    int y;
} Point;
```

When run, your `main()` function should use the templated `List` class you've written yourself to produce this output:

```
0 100 200 300 400 500 600 700 800 900
(0, 0) (10, 100) (20, 200) (30, 300) (40, 400)
```

Naturally, this output should be generated by accessing the members of your templated `List` class, not by a hard-coded print statement.

To get you started, we've written a non-templated `IntList` class that just handles lists of integers:

```
class IntList {
    int * list;

public:
    int length;

    IntList(int len) {
        list = new int[len];
        length = len;
    }

    ~IntList() {
        delete[] list;
    }

    int get(int index) {
        return list[index];
    }

    void set(int index, int val) {
        list[index] = val;
    }
};
```

You should use this class as a model for your own, templated `List` class, but you won't need `IntList` at all in the final code that you turn in, because you will have replaced it with your templated `List` class.

Make sure your program compiles without warning, runs, and *definitely* use `valgrind` to ensure you have no memory leaks.

```
$ g++ -Wall p2_templates.cpp -o p2_templates
$ ./p2_templates
<your test output>
```

Solutions

Solutions are not available for this assignment.



Over 2,500 courses & materials

Freely sharing knowledge with learners and educators around the world. [Learn more](#)

[Accessibility](#)

[Creative Commons License](#)

[Terms and Conditions](#)

Proud member of:  The logo for Open Education GLOBAL, featuring a blue circular icon with a grid pattern next to the text 'Open Education GLOBAL'.

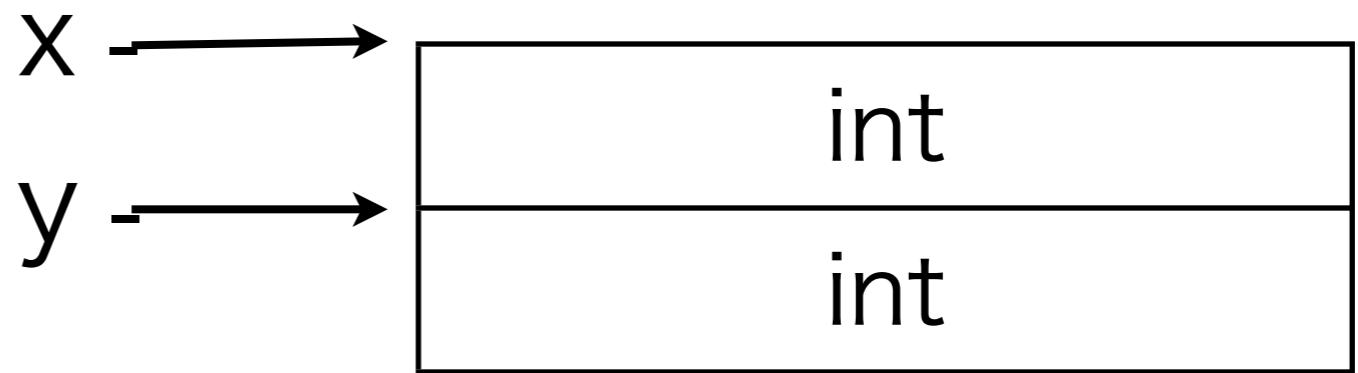


© 2001–2025 Massachusetts Institute of Technology

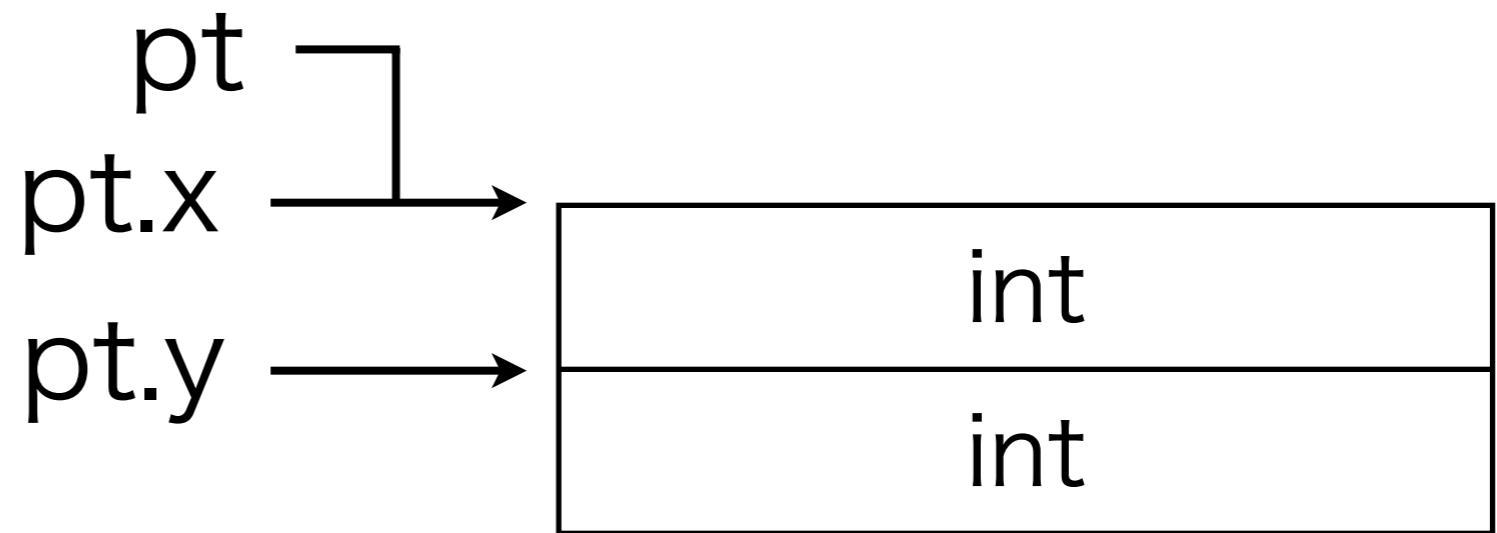
C++ Inheritance

Bits & Bobs

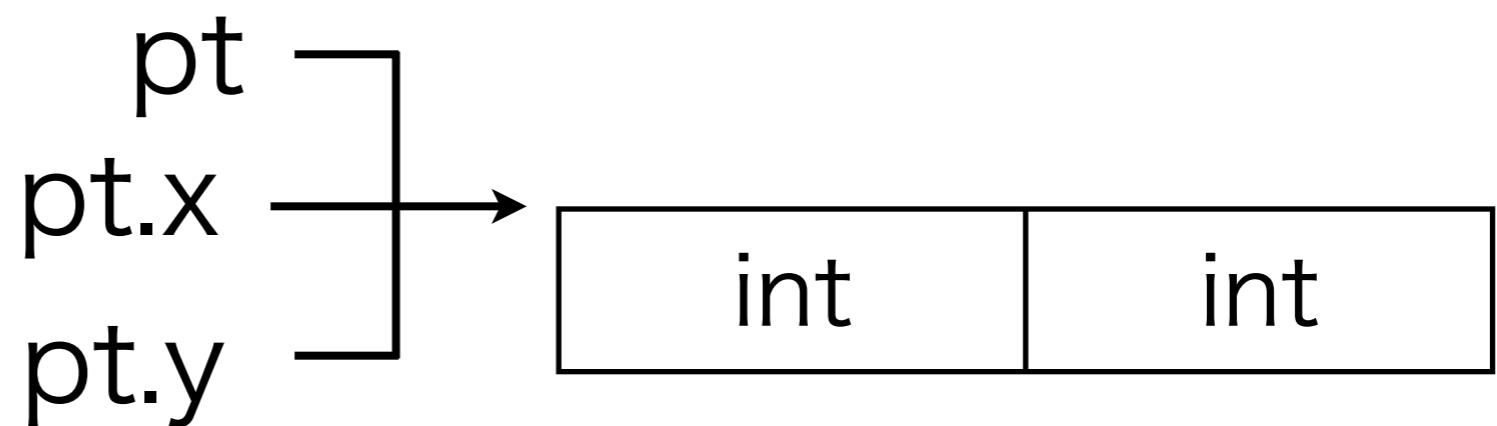
```
int x;  
int y;
```



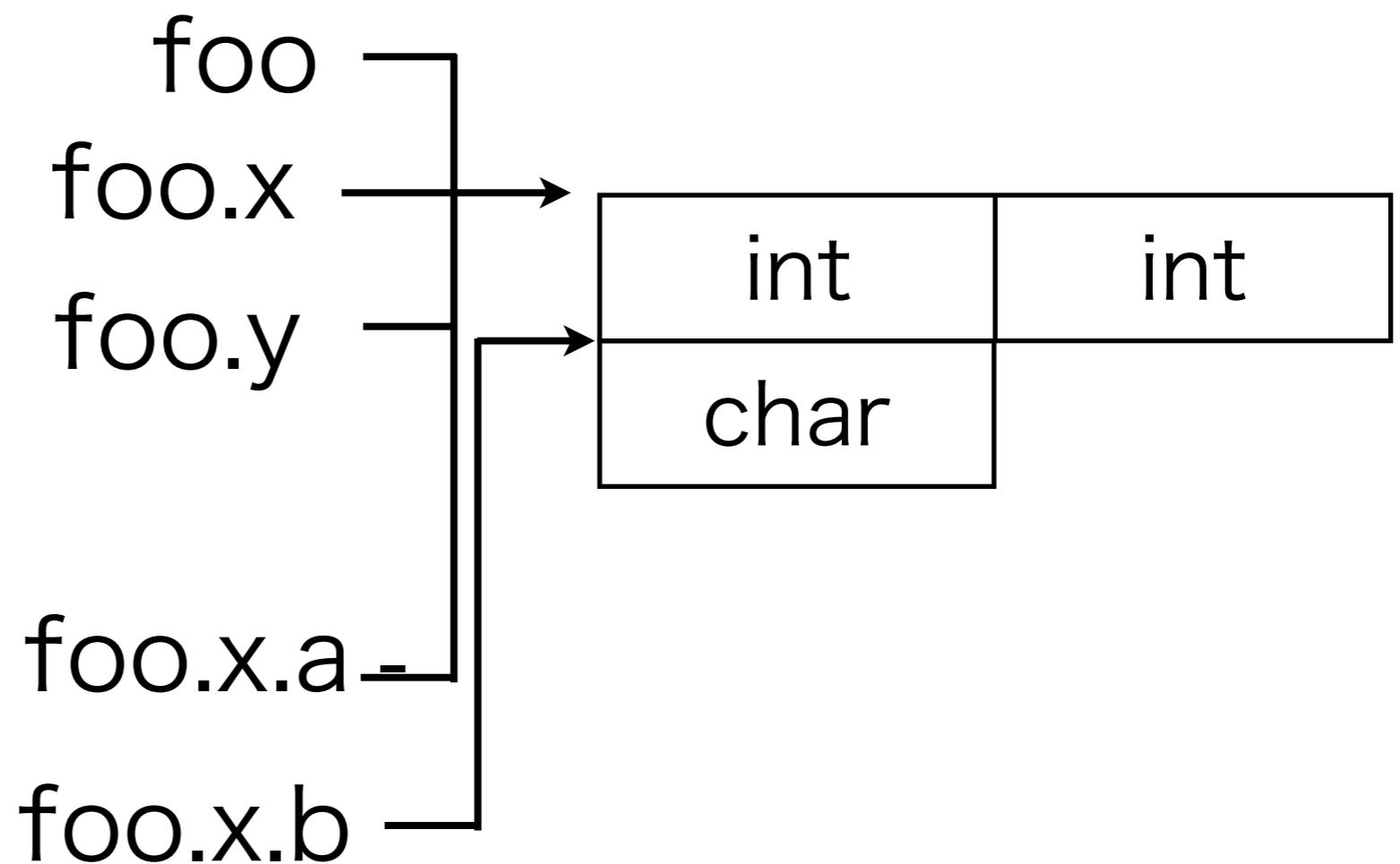
```
struct {  
    int x;  
    int y;  
} pt;
```



```
union {  
    int x;  
    int y;  
} pt;
```



```
union {  
    struct {  
        int a;  
        char b;  
    } x;  
    int y;  
} foo;
```



Why Inheritance?

```
struct Circle {      struct Square {  
    int x, y;  
    int radius;  
    void draw();  
};  
};
```

```
struct Circle {      struct Square {  
    int x, y;  
    int radius;  
    void draw();  
};  
};
```

```
Circle *circles[nc];  
Square *squares[ns];
```

```
struct Circle {      struct Square {  
    int x, y;  
    int radius;  
    void draw();  
};  
};
```

```
Circle *circles[nc];  
Square *squares[ns];
```

```
for(int i = 0; i < nc; i++)  
    circles[i].draw();  
for(int i = 0; i < ns; i++)  
    squares[i].draw();
```

```
Circle *circles[nc];
```

```
Square *squares[ns];
```

```
for(int i = 0; i < nc; i++)  
    circles[i].draw();
```

```
for(int i = 0; i < ns; i++)  
    squares[i].draw();
```

```
for(int i = 0; i < nc; i++)
```

```
    delete circles[i];
```

```
for(int i = 0; i < ns; i++)
```

```
    delete squares[i];
```

```
for(int i = 0; i < nc; i++)
```

```
    printf("o%d\n", circles[i].width);
```

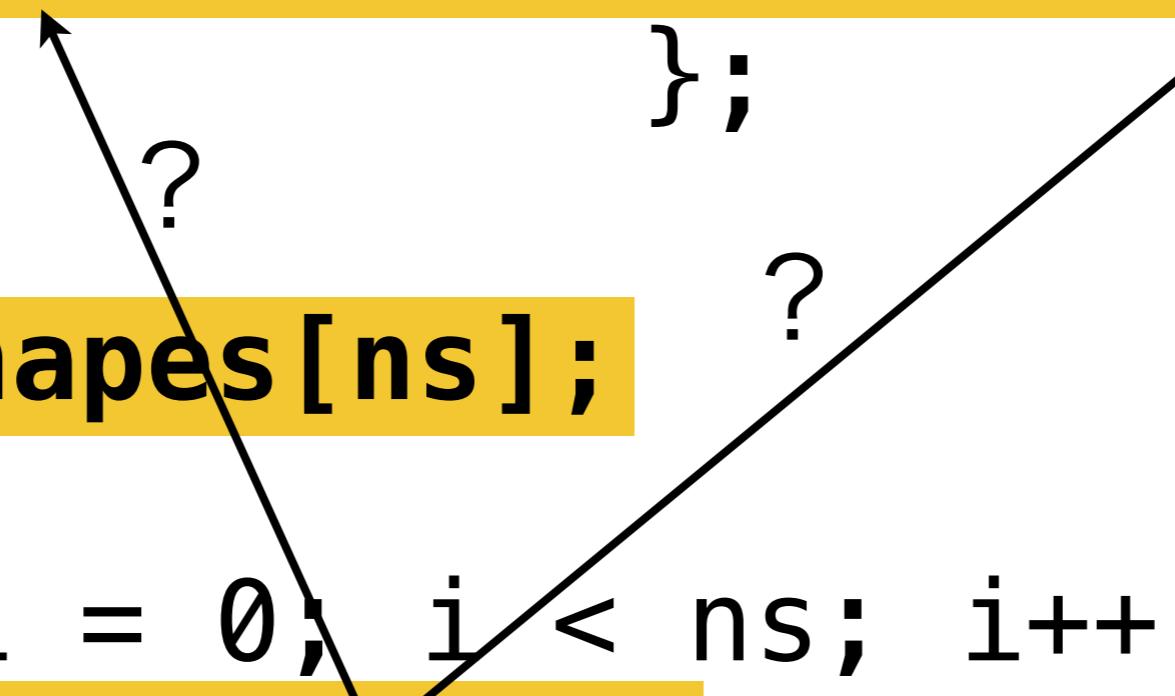
```
struct Circle {  
    int x, y;  
    int radius;  
    void draw();  
};
```

```
struct Square {  
    int x, y;  
    int width;  
    void draw();  
};
```

```
Shape *shapes[ns];
```

```
for(int i = 0; i < ns; i++)  
    shapes[i].draw();
```

```
for(int i = 0; i < ns; i++)  
    delete shapes[i];
```



Inheritance

```
class Shape {  
public:  
    virtual void draw() = 0;  
};
```

```
class Circle : public Shape {  
private:  
    int x, y;  
    int radius;  
public:  
    virtual void draw();  
};
```

```
void Circle::draw() {  
    ...  
}
```

Best Practice

- 1) Subclassing
- 2) virtual

- 1) Subclassing
- 2) virtual

1) Subclassing

inherit behavior from the parent

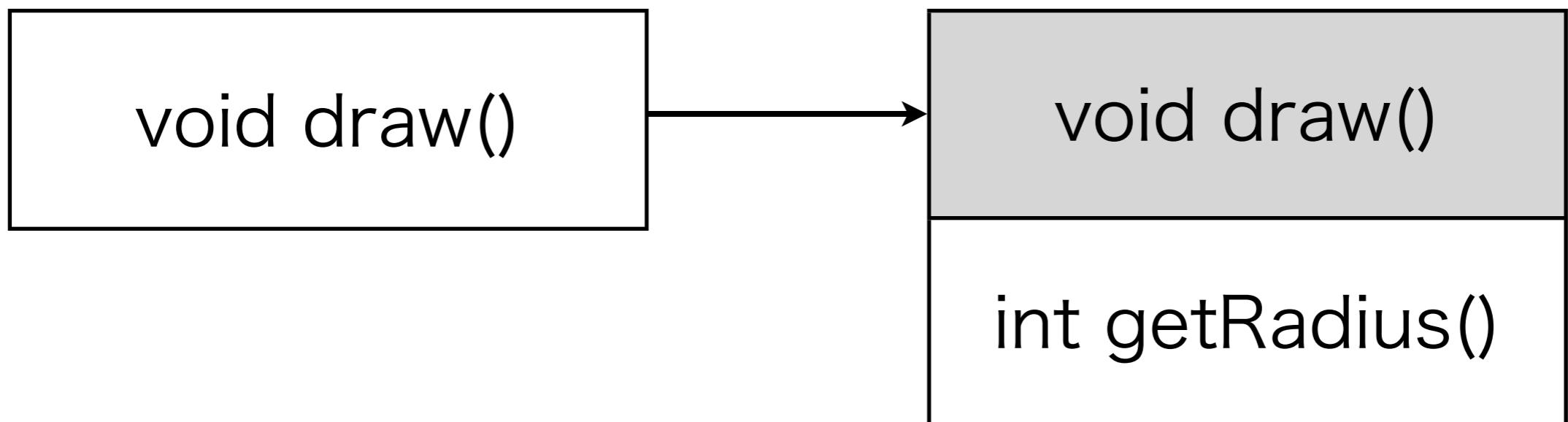
```
class Shape {  
public:  
    void draw();  
};
```

```
class Circle : public Shape {  
public:  
    int getRadius();  
};
```

```
int main() {  
    Circle circle;  
    circle.draw();  
}
```

Shape

Circle : public Shape



1) Subclassing

inherit fields from the parent

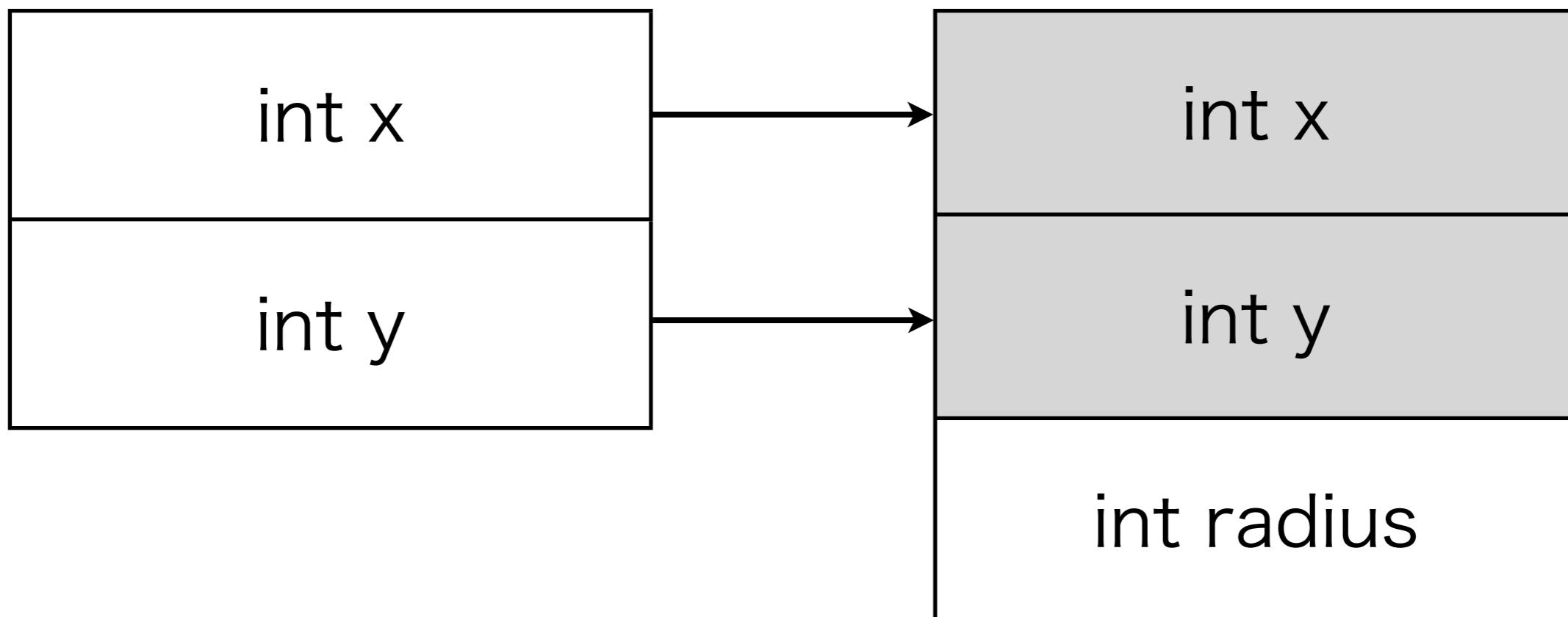
```
class Shape {  
public:  
    int x, y;  
};
```

```
class Circle : public Shape {  
public:  
    int radius;  
};
```

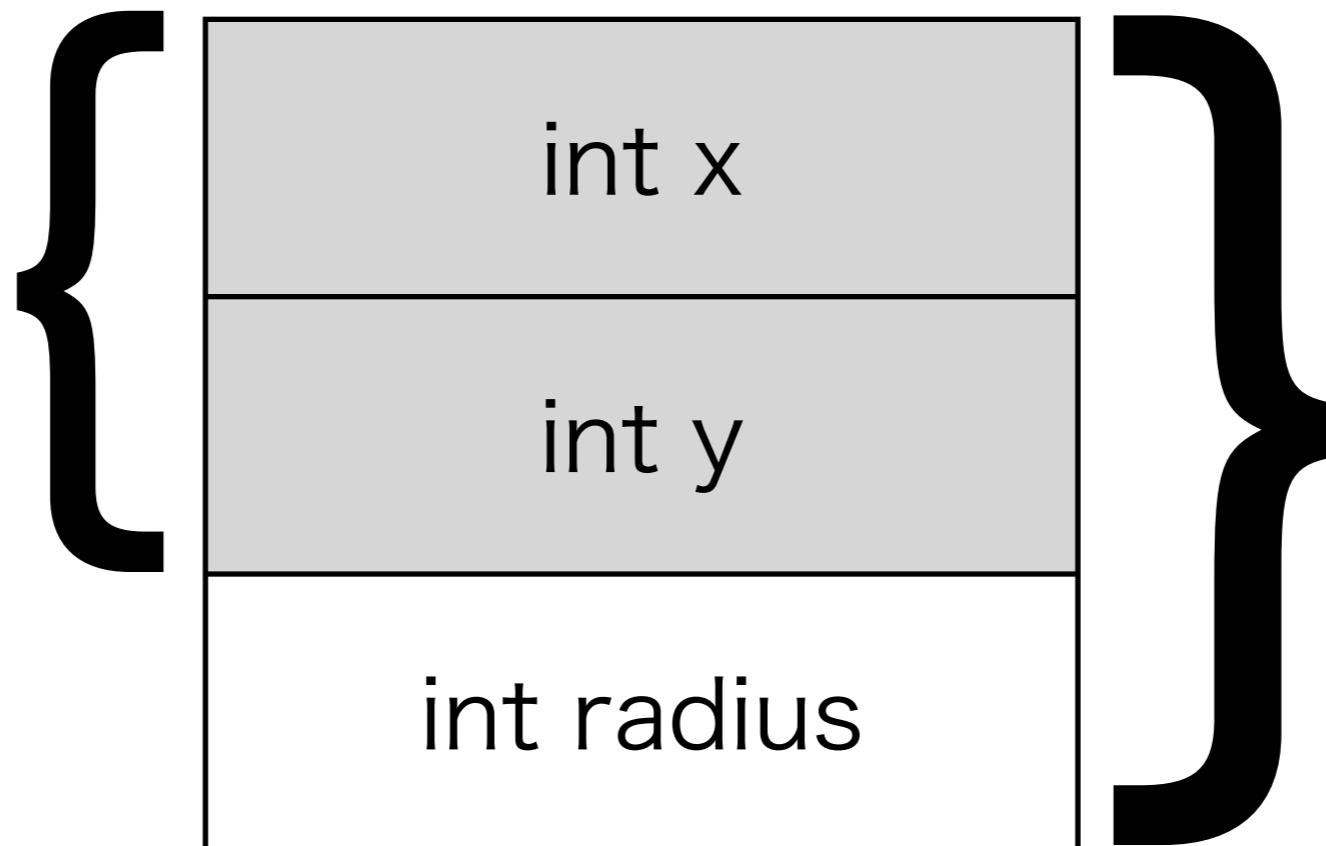
```
int main() {  
    Circle circle;  
    circle.x = 5;  
}
```

Shape

Circle : public Shape



Shape



Circle

is-a or has-a?

```
class Circle  
  : public Shape {  
public:  
  int radius;  
};
```

```
class Circle {  
public:  
  Shape shape;  
  int radius;  
};
```

```
class Circle  
  : public Shape {  
public:  
  int radius;  
};
```

circle.x;

```
class Circle {  
public:  
  Shape shape;  
  int radius;  
};
```

circle.shape.x;

1) Subclassing

public/protected/private fields

```
class Shape {  
public:  
    int x;  
private: ← only accessible in  
    int y;  
};  
Shape class
```

```
void Circle::foo() {  
    printf("%d", x);  
    printf("%d", y); // compile error  
}
```

```
class Shape {  
public:  
    int x;  
protected: ← accessible in Shape class  
    int y;  
};  
                                         and in subclasses
```

```
void Circle::foo() {  
    printf("%d", x);  
    printf("%d", y);  
}
```

```
int main() {  
    Circle circle;  
    circle.x = 0; // compile error  
}
```

1) Subclassing

public/protected/private inheritance

```
class Shape {  
public:  
    void draw();  
};
```

```
class Circle : public Shape {  
};
```

```
class Shape {  
public:  
    void draw();  
};
```

```
class Circle : protected Shape {  
};
```

```
class Shape {  
public:  
    void draw();  
};
```

```
class Circle : protected Shape {  
protected:  
    int getRadius();  
};
```

The **inheritance** is protected.

If you can access **getRadius()**,
you can access **draw()**

```
class Shape {  
public:  
    void draw();  
};
```

```
class Circle : private Shape {  
private:  
    int getRadius();  
};
```

The **inheritance** is private.

If you can access **getRadius()**,
you can access **draw()**

private inheritance:
is-a or has-a

1) Subclassing

multiple inheritance

```
class Color {  
public: virtual void print();  
};
```

```
class Mood {  
public: virtual void print();  
};
```

```
class Blue : public Color, public Mood {  
public:  
    virtual void print() {  
        this->Color::print();  
        this->Mood::print();  
    }  
};
```

1) Subclassing

slicing

```
struct Cow {  
    void speak() {  
        printf("Moo.\n");  
    }  
};  
  
struct Werecow : public Cow {  
    bool transformed;  
    void speak() {  
        if (transformed)  
            printf("Aoooooh!\n");  
        else  
            printf("Moo.\n");  
    }  
};
```

```
Werewolf wcow;
wcow.transformed = true;

Cow cows[2];
cows[0] = Cow();
cows[1] = wcow;

for (int i = 0; i < 2; i++)
    cows[i].speak();
wcow.speak();

// Output:
//      Moo.
//      Moo.
//      Aaaaaah!
```

```
void poke(Cow cow) {  
    cow.speak();  
}
```

```
Cow judy;  
Werecow bev;  
bev.transformed = true;
```

```
poke(judy);  
poke(bev);  
bev.speak();
```

```
// Output:  
//      Moo.  
//      Moo.  
//      Aoooooh!
```

Cow



Werecow

bool transformed

Use pointers
Use virtual

- 1) Subclassing
- 2) `virtual`

```
class Shape {  
public:  
    void draw() { printf("shape\n"); }  
};
```

```
class Circle : public Shape {  
public:  
    void draw() { printf("circle\n"); }  
};
```

```
class Shape {  
public:  
    void draw() { printf("shape\n"); }  
};
```

```
class Circle : public Shape {  
public:  
    void draw() { printf("circle\n"); }  
};
```

```
Circle *circle = new Circle;  
circle->draw(); // "circle"
```

```
class Shape {  
public:  
    void draw() { printf("shape\n"); }  
};
```

```
class Circle : public Shape {  
public:  
    void draw() { printf("circle\n"); }  
};
```

```
Shape *shape = new Circle;  
shape->draw(); // "shape"
```

```
class Shape {  
public:  
    void draw() { printf("shape\n"); }  
};
```

```
class Circle : public Shape {  
public:  
    void draw() { printf("circle\n"); }  
};
```

```
Shape *shape = new Circle;  
shape->draw(); // "shape"
```

draw() is **non-virtual**,
so it's compiled like a C call

Non-virtual functions are -
determined at compile-time -

```
class Cat {  
public:  
    void yawn(int duration);  
};
```

```
Cat cat, *pcat = new SuperCat;  
cat.yawn(4);  
pcat->yawn(4);
```

Both use Cat::yawn -
because both have type Cat -

Virtual functions are
determined at run-time

```
class Cat {  
public:  
    virtual void yawn(int duration);  
};
```

```
Cat cat, *pcat = new SuperCat;  
cat.yawn(4);  
pcat->yawn(4);
```

Use Cat::yawn and SuperCat::yawn
(pcat's type is checked every time it's called)

non-virtual: compile-time
virtual: run-time

2) virtual
pure virtual methods

```
class Shape {  
public:  
    virtual void draw() = 0;  
};
```

```
int main() {  
    Shape shape;  
}
```

pure.cpp: In function ‘int main()’:

pure.cpp:7: error: cannot declare variable ‘shape’ to be of abstract type ‘Shape’
pure.cpp:1: note: because the following virtual functions are pure within ‘Shape’:
pure.cpp:3: note: virtual void Shape::draw()

```
class Drawable {  
public:  
    virtual void draw() = 0;  
};  
  
class Fish : public Drawable {  
public:  
    virtual void draw();  
};  
  
int main() {  
    Drawable *drawables[3];  
    drawables[0] = new Fish;  
    drawables[1] = new Salami;  
    drawables[2] = new JackSparrow;  
}
```

2) virtual

destructors

Make virtual destructors -

```
class Fish {  
public:  
    Fish() {  
        gills[0] = new Gill;  
        gills[1] = new Gill;  
    }  
    virtual ~Fish() {  
        delete gills[0];  
        delete gills[1];  
    }  
private:  
    Gill *gills[2];  
};
```

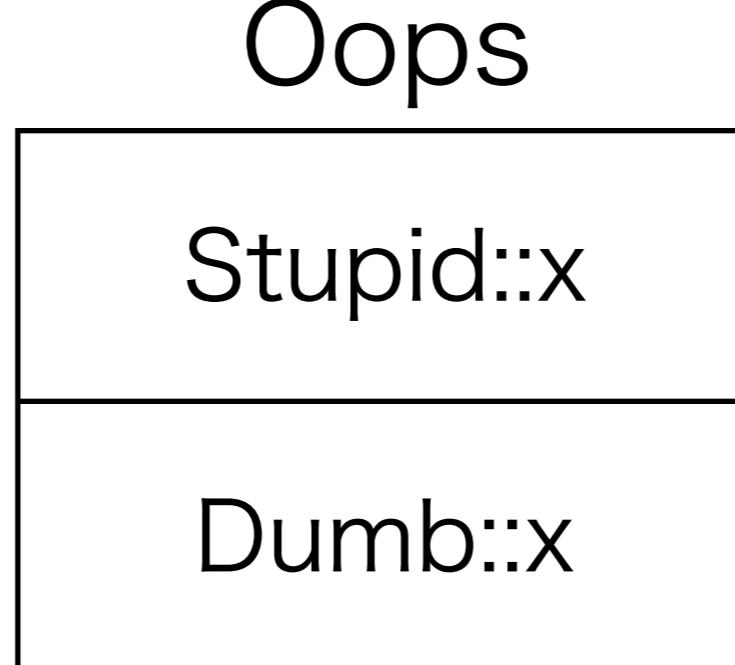
2) virtual

virtual inheritance

```
class Goofball {  
    int x;  
};
```

```
class Stupid : public Goofball { };  
class Dumb : public Goofball { };
```

```
class Oops : public Stupid, public Dumb {  
};
```



```
class Goofball {  
    int x;  
};  
  
class Stupid : public Goofball { };  
class Dumb : public Goofball { };  
  
class Oops : public Stupid, public Dumb {  
    int fail();  
};  
  
int Oops::fail() {  
    Stupid::x = 1; Dumb::x = 2;  
    return Stupid::x + Dumb::x; // 3  
}
```

```
class Goofball {  
    int x;  
};
```

```
class Stupid : virtual public Goofball { };  
class Dumb : virtual public Goofball { };
```

```
class Oops : public Stupid, public Dumb {  
    int fail();  
};
```

```
int Oops::fail() {  
    Stupid::x = 1; Dumb::x = 2;  
    return Stupid::x + Dumb::x; // 4  
}
```

Conclusion

```
class Shape {  
public:  
    virtual void draw() = 0;  
};
```

```
class Circle : public Shape {  
private:  
    int x, y;  
    int radius;  
public:  
    virtual void draw();  
};
```

```
void Circle::draw() {  
    ...  
}
```

Best Practice

MIT OpenCourseWare
<http://ocw.mit.edu>

6.S096 Introduction to C and C++
IAP 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.



OpenCourseWare

6.S096 | January IAP 2013 | Undergraduate

Introduction To C And C++

[Menu](#) [More Info](#) [Feedback](#)

Lectures and Assignments

C++ Inheritance

Lecture Notes

[Lecture 6: C++ Inheritance \(PDF\)](#)

Lab Exercises

Take a look at this example code:

```
#include <stdio.h>

class Shape {
public:
    virtual ~Shape();
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    virtual ~Circle();
    virtual void draw();
};

Shape::~Shape() {
    printf("shape destructor\n");
}

// void Shape::draw() {
//     printf("Shape::draw\n");
// }

Circle::~Circle() {
    printf("circle destructor\n");
}
```

```
void Circle::draw() {
    printf("Circle::draw\n");
}

int main() {
    Shape *shape = new Circle;
    shape->draw();
    delete shape;

    return 0;
}
```

Put it in a file named `lab6.cpp` and then compile it like this:

```
$ g++ -Wall lab6.cpp -o lab6
$ ./lab6
Circle::draw
circle destructor
shape destructor
```

Verify your understanding of how the `virtual` keyword and method overriding work by performing a few experiments:

1. Remove the `virtual` keyword from each location individually, recompiling and running each time to see how the output changes. Can you predict what will and will not work?
2. Try making `Shape::draw` non-pure by removing `= 0` from its declaration.
3. Try changing `shape` (in `main()`) from a pointer to a stack-allocated variable.

Assignment 6

[rps \(CPP\)](#)

In the file `rps.cpp`, implement a class called `Tool`. It should have an `int` field called `strength` and a `char` field called `type`. You may make them either private or protected. The `Tool` class should also contain the function `void setStrength(int)`, which sets the strength for the `Tool`.

Create 3 more classes called `Rock`, `Paper`, and `Scissors`, which inherit from `Tool`. Each of these classes will need a constructor which will take in an `int` that is used to initialize the `strength` field. The constructor should also initialize the `type` field using '`r`' for `Rock`, '`p`' for `Paper`, and '`s`' for `Scissors`.

These classes will also need a public function `bool fight(Tool)` that compares their strengths in the following way:

- Rock's strength is doubled (temporarily) when fighting scissors, but halved (temporarily) when fighting paper.

- In the same way, paper has the advantage against rock, and scissors against paper.
- The **strength** field shouldn't change in the function, which returns **true** if the original class wins in strength and **false** otherwise.

You may also include any extra auxiliary functions and/or fields in any of these classes. Run the program without changing the main function, and verify that the results are correct.

```
$ g++ -Wall rps.cpp -o rps
$ ./rps
<your test output>
```

Solutions

Solutions are not available for this assignment.



Over 2,500 courses & materials

Freely sharing knowledge with learners and educators around the world. [Learn more](#)

[Accessibility](#)

[Creative Commons License](#)

[Terms and Conditions](#)

Proud member of: The logo for Open Education GLOBAL, featuring a blue circular icon with a stylized 'G' or globe design, followed by the text 'Open Education GLOBAL'.



© 2001–2025 Massachusetts Institute of Technology

C++ (& C) Grab Bag

Final Project: Due in 2 Days
Complete Something

Parent destructors

```
struct Buffer {  
    Buffer(int s) { buf = new char[s]; }  
    ~Buffer() { delete [] buf; }  
    char *buf;  
};
```

```
struct FBuffer : public Buffer {  
    FBuffer(int s) : Buffer(s) {  
        f = fopen("file", "w");  
    }  
    ~FBuffer() { fclose(f); }  
    void write() { fwrite(buf, 1, 40, f); }  
    FILE *f;  
};
```

```
struct Buffer {  
    Buffer(int s);  
    ~Buffer();  
    char *buf;  
};
```

```
struct FBuffer  
: public Buffer {  
    FBuffer(int s);  
    ~FBuffer();  
    void write();  
    FILE *f;  
};
```

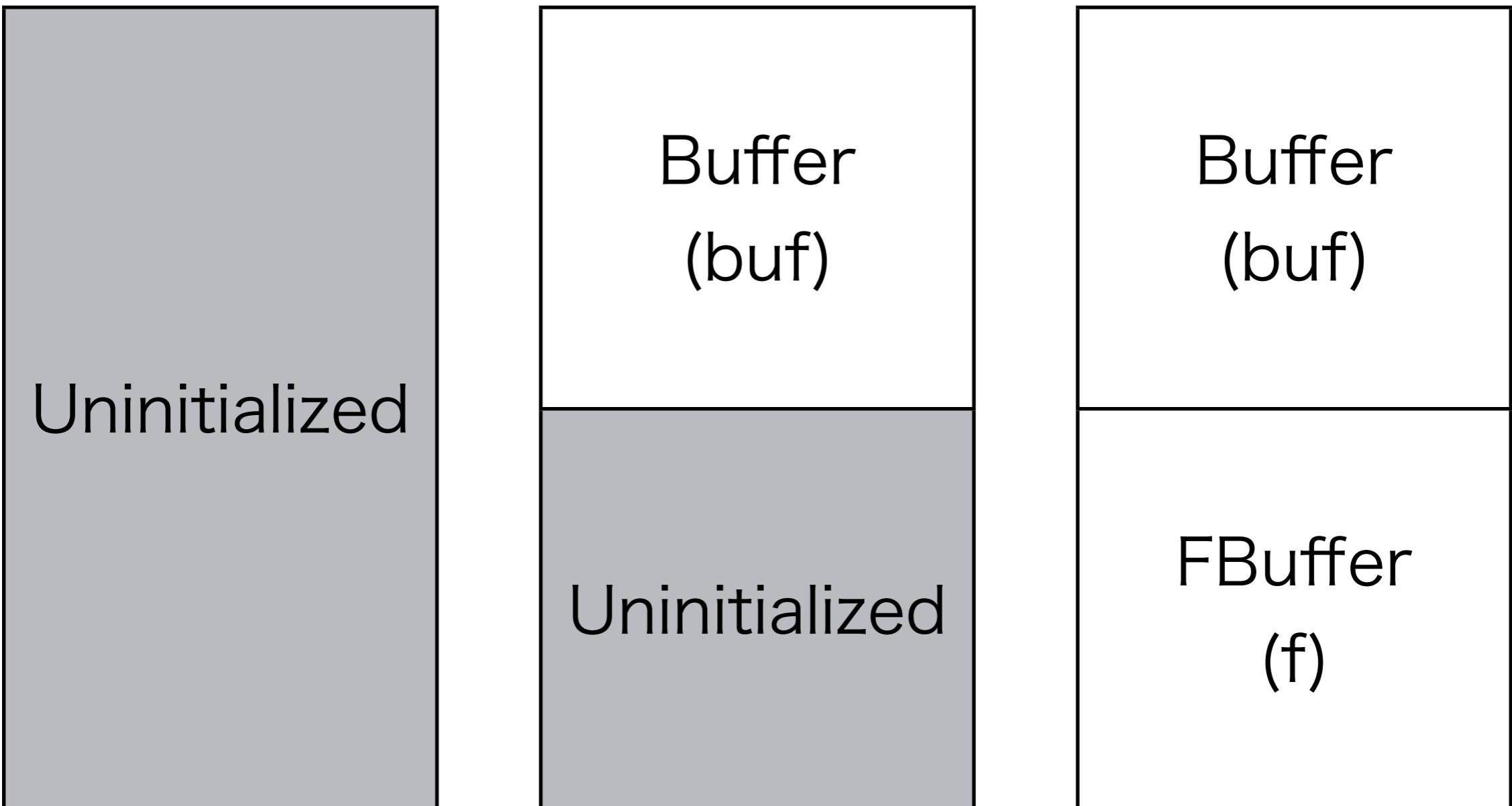
```
Buffer *buf = new Buffer(128);  
delete buf;  
// ✓
```

```
struct Buffer {  
    Buffer(int s);  
    ~Buffer();  
    char *buf;  
};
```

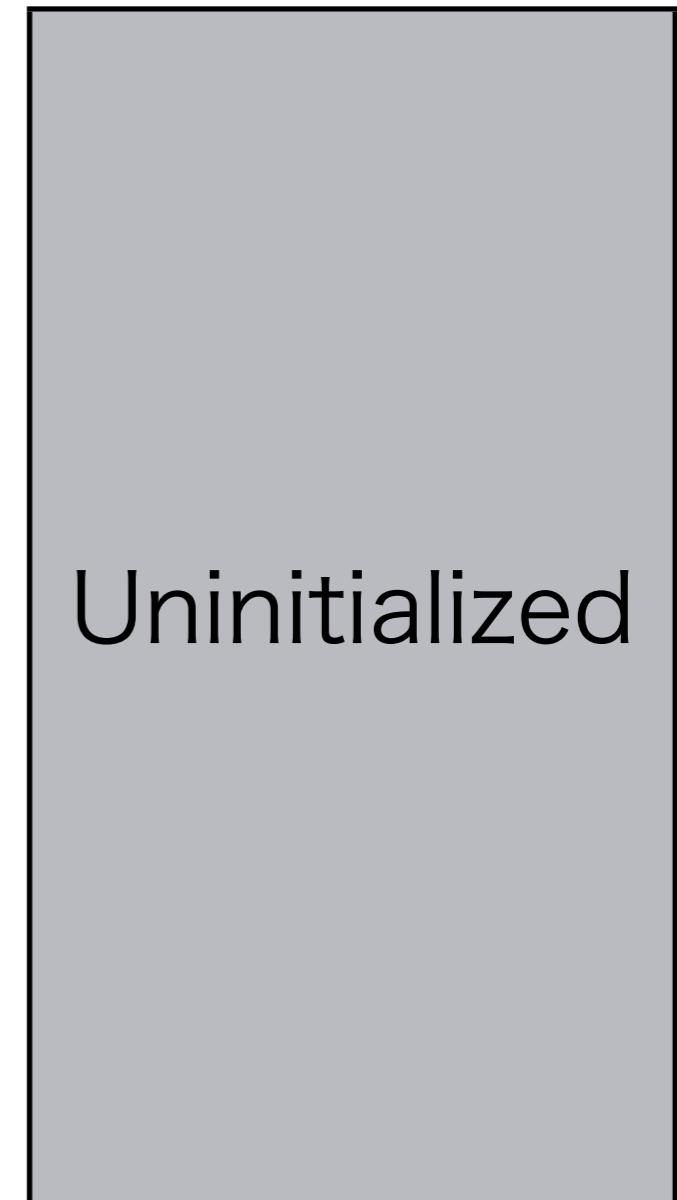
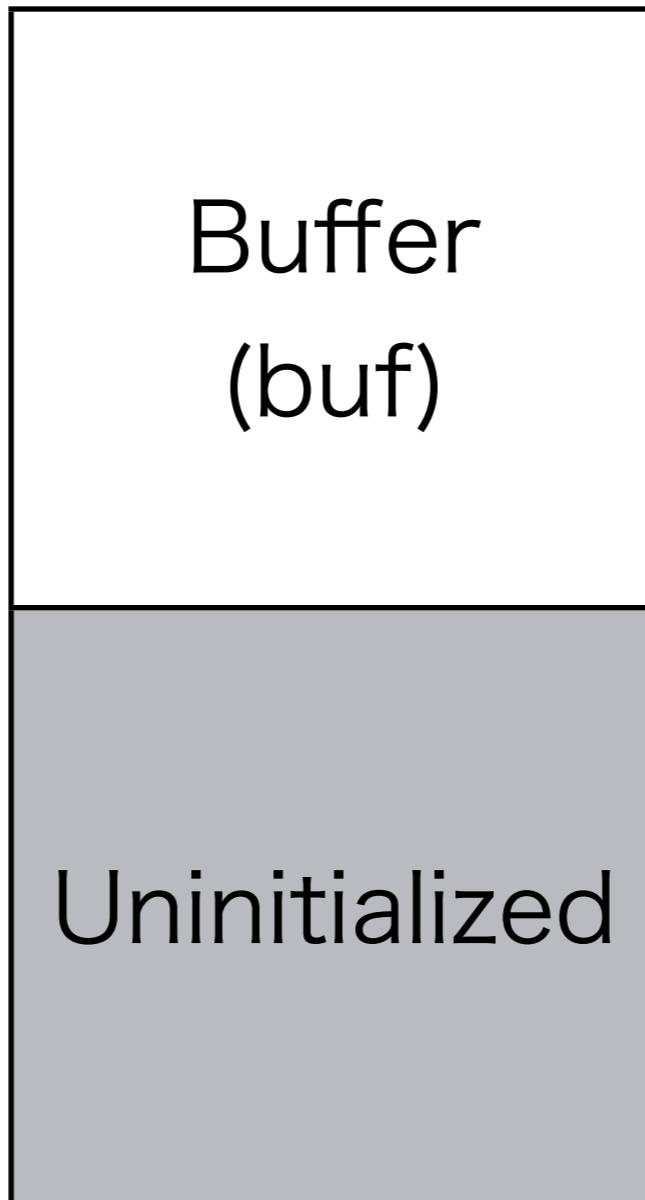
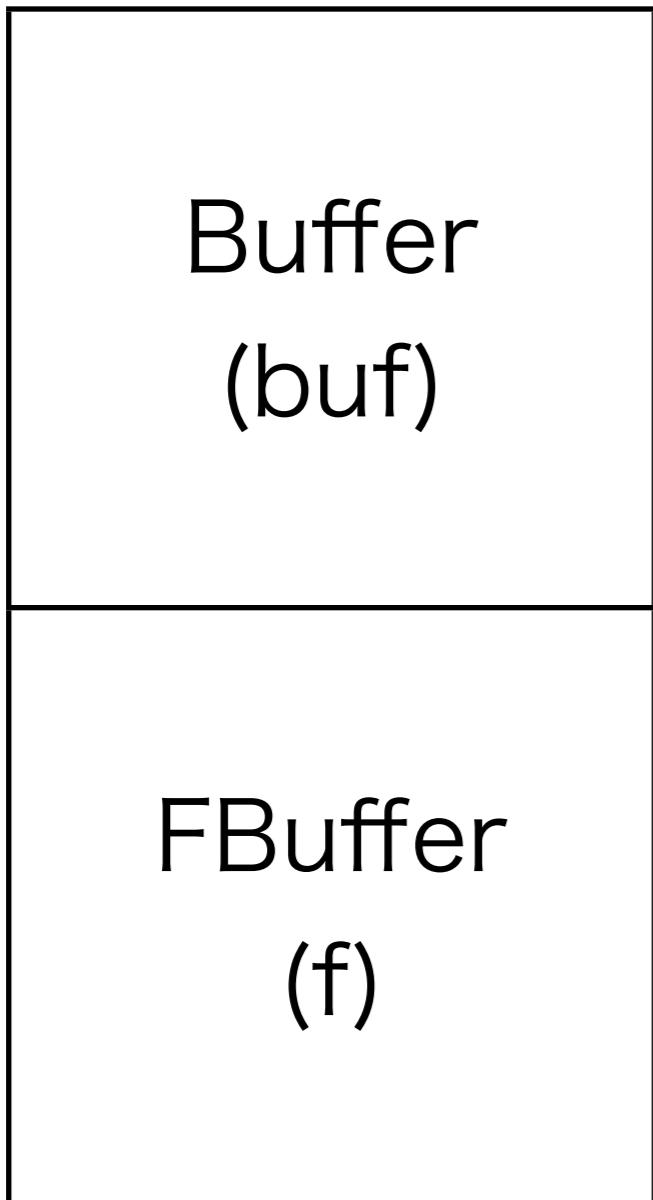
```
struct FBuffer  
: public Buffer {  
    FBuffer(int s);  
    ~FBuffer();  
    void write();  
    FILE *f;  
};
```

```
FBuffer *fbuf = new FBuffer(128);  
delete fbuf;  
// ✓
```

Construction



Destruction



```
struct Buffer {  
    Buffer(int s);  
    ~Buffer();  
    char *buf;  
};
```

```
struct FBuffer  
: public Buffer {  
    FBuffer(int s);  
    ~FBuffer();  
    void write();  
    FILE *f;  
};
```

```
Buffer *fbuf = new FBuffer(128);  
delete fbuf;  
// ✗ only ~Buffer is called
```

```
struct Buffer {           struct FBuffer
    Buffer();           : public Buffer {
    ~Buffer();          FBuffer();
    virtual ~Buffer();  virtual ~FBuffer();
    char *buf;          void write();
};                      FILE *f;
};
```

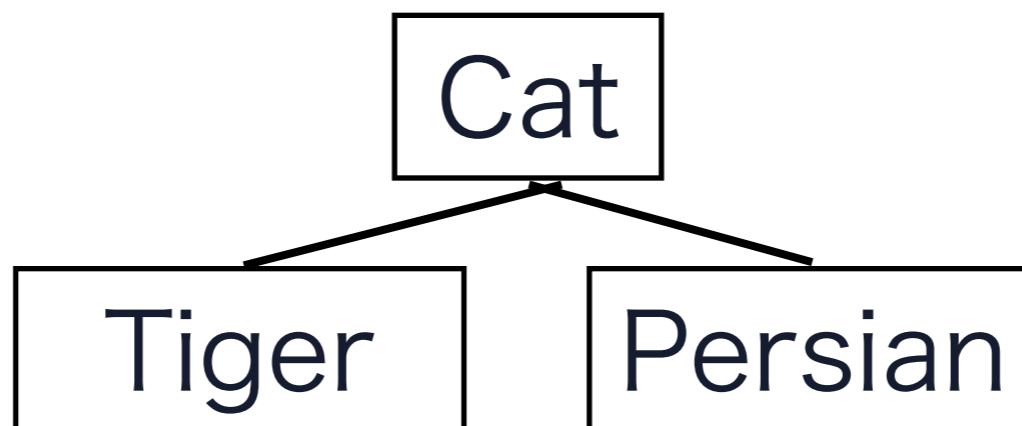
```
Buffer *fbuf = new FBuffer;
delete fbuf;
// ✓
```

C++ Casts

```
// C cast  
char *buf = (char *)malloc(128);
```

```
// C-style cast  
float b = 98.6;  
int a = int(b);
```

```
// C-style cat casts
class Cat { };
class Tiger : public Cat { };
class Persian : public Cat { };
Cat *c = new Persian;
Tiger *t = (Tiger *)c; // whoops!
```



```
// valid up-cast
```

```
Tiger *t = new Tiger;
```

```
Cat *c1 = (Cat *)t;
```

```
Cat *c2 = static_cast<Cat *>(t);
```

```
Cat *c3 = dynamic_cast<Cat *>(t);
```

```
// almost valid down-cast
```

```
Cat *c = new Tiger;
```

```
Tiger *t1 = (Tiger *)c;
```

```
Tiger *t2 = static_cast<Tiger *>(c);
```

```
Tiger *t3 = dynamic_cast<Tiger *>(c);
```

```
// compile error
```

```
// valid down-cast
```

```
class Cat { virtual void purr() { }  };
class Tiger : public Cat { };
class Persian : public Cat { };
```

```
Cat *c = new Tiger;
```

```
Tiger *t1 = (Tiger *)c;
Tiger *t2 = static_cast<Tiger *>(c);
Tiger *t3 = dynamic_cast<Tiger *>(c);
```

```
// invalid down-cast
```

```
Cat *c = new Persian;
```

```
Tiger *t1 = (Tiger *)c;
```

```
Tiger *t2 = static_cast<Tiger *>(c);
```

```
Tiger *t3 = dynamic_cast<Tiger *>(c);
```

```
// t1 & t2 are invalid pointers
```

```
// t3 is NULL
```

References

```
void swap(int *a, int *b) {  
    int tmp = *a;  
    *a = *b;  
    *b = tmp;  
}  
  
int main() {  
    int x = 2, y = 3;  
    swap(&x, &y);  
}
```

```
void swap(int &a, int &b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main() {  
    int x = 2, y = 3;  
    swap(x, y);  
}
```

Hello, World!

```
#include <iostream>

int main() {
    std::cout << "Hello, World!"
              << std::endl;
    return 0;
}
```

```
#include <iostream>

int main() {
    std::cout << "Hello, World!"
                << std::endl;
    return 0;
}
```

Namespaces

```
SNDFILE *open(const char *);  
count_t seek(SNDFILE *, count_t);  
int error(SNDFILE *);
```

```
SNDFILE *sf_open(const char *);  
count_t sf_seek(SNDFILE *, count_t);  
int sf_error(SNDFILE *);
```

```
namespace sf {  
    SNDFILE *open(const char *);  
    count_t seek(SNDFILE *, count_t);  
    int error(SNDFILE *);  
}
```

```
#include <iostream>

int main() {
    std::cout << "Hello, World!"
                << std::endl;
    return 0;
}
```

```
std::cout << "Hello, World!"  
        << std::endl;
```

```
using namespace std;  
cout << "Hello, World!"  
    << endl;
```

```
using std::cout;  
using std::endl;  
cout << "Hello, World!"  
    << endl;
```

```
g++ -E hello.cpp
```

iostream

```
namespace std {  
    extern istream cin;  
    extern ostream cout;  
    extern ostream cerr;  
}
```

```
namespace std {  
    extern istream cin;  
    extern ostream cout;  
    extern ostream cerr;  
  
    class ActionLawsuit {  
    };  
}
```

```
extern istream cin;
extern ostream cout;
extern ostream cerr;

class ActionLawsuit {
};
```

```
namespace super {  
    namespace std {  
        extern istream cin;  
        extern ostream cout;  
        extern ostream cerr;  
  
        class ActionLawsuit {  
        };  
    }  
}  
  
super::std::ActionLawsuit;
```

extern

iostream

```
namespace std {  
    extern istream cin;  
    extern ostream cout;  
    extern ostream cerr;  
}
```

iostream

ostream cout;

main.c

```
#include <iostream>

int main() {
    cout << "i";
    foo();
}
```

foo.c

```
#include <iostream>

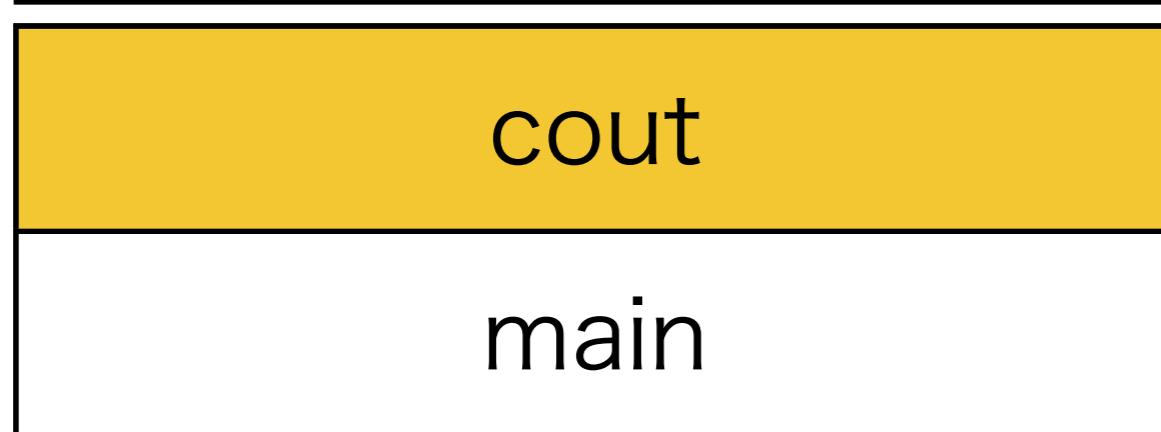
int foo() {
    cout << "Phone";
}
```

main.c (preprocessed)

```
ostream cout;
```

```
int main() {  
    cout << "i";  
    foo();  
}
```

main.o



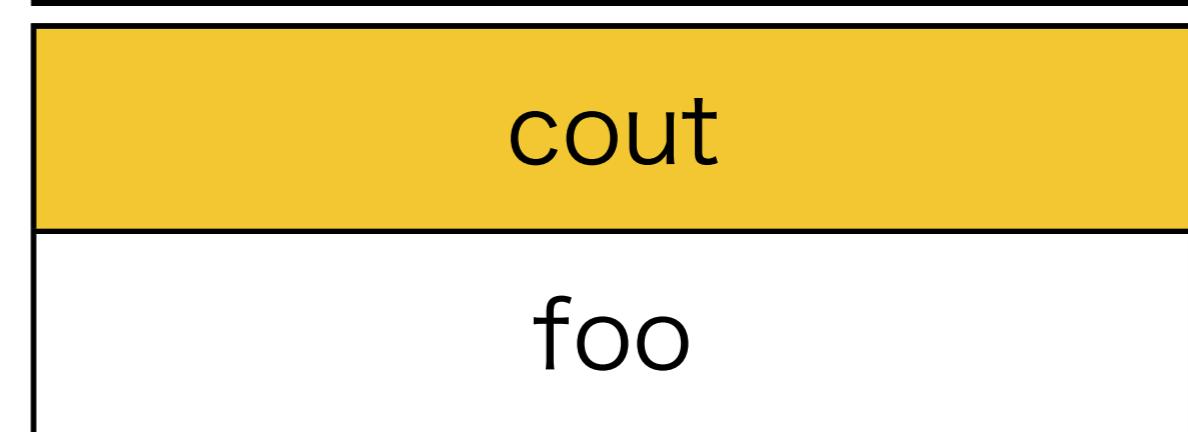
ld: 1 duplicate symbol for architecture x86_64

foo.c (preprocessed)

```
ostream cout;
```

```
int foo() {  
    cout << "Phone";  
}
```

foo.o



main.c (preprocessed)

```
extern ostream cout; extern ostream cout;
```

```
int main() {  
    cout << "i";  
    foo();  
}
```

```
int foo() {  
    cout << "Phone";  
}
```

main.o

```
main
```

foo.o

```
foo
```

<standard library>

```
cout
```

```
#include <iostream>

int main() {
    std::cout << "Hello, World!"  
        << std::endl;
    return 0;
}
```

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    int a = 2 << 1;
    return 0;
}
```

Operator Overloading

```
struct vec2 {  
    vec2(float x, float y)  
        : x(x), y(y) {}  
    float x, y;  
};  
  
int main() {  
    vec2 a(1, 0);  
    vec2 b(1, 3);  
    vec2 c = a + b; // compile error  
}
```

```
vec.cpp: In function ‘int main()’:  
vec.cpp:12: error: no match for ‘operator+’ in ‘a + b’
```

```
vec2 vec2::add(const vec2 &o) {  
    return vec2(x + o.x, y + o.y);  
}
```

```
int main() {  
    vec2 a(1, 0), b(1, 3);  
    vec2 c = a.add(b);  
}
```

```
vec2 vec2::operator +(const vec2 &o) {  
    return vec2(x + o.x, y + o.y);  
}
```

```
int main() {  
    vec2 a(1, 0), b(1, 3);  
    vec2 c = a + b;  
    vec2 d = a.operator+(b);  
}
```

```
vec2 operator +(vec2 &v, const vec2 &o) {  
    return vec2(x + o.x, y + o.y);  
}
```

```
int main() {  
    vec2 a(1, 0), b(1, 3);  
    vec2 c = a + b;  
}
```

a + b	a != b
a - b	a && b
a * b	a b
a / b	a & b
a % b	a b
a < b	a ^ b
a <= b	a << b
a == b	a >> b
a >= b	a, b
a > b	a[b]

vec2 operator+(const vec2 &o);

+a

-a

++a

a++

--a

a--

! a

~a

*a

&a

vec2 operator+();

a = b	a &= b
a += b	a = b
a -= b	a ^= b
a *= b	a <<= b
a /= b	a >>= b
a %= b	a = (b += c)

```
vec2 &vec2::operator+=(const vec2 &o)
{
    x += o.x;
    y += o.y;
    return *this;
}
```

(**this** is a pointer
to the object)

Streams

```
struct Foo {  
    char *str() const {  
        return "Foo!";  
    }  
};  
  
ostream &  
operator<<(ostream &os, const Foo &f) {  
    return os << f.str();  
}  
  
int main() {  
    Foo f;  
    std::cout << f << std::endl;  
}
```

MIT OpenCourseWare
<http://ocw.mit.edu>

6.S096 Introduction to C and C++
IAP 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.

6.S096: Introduction to C/C++

Frank Li, Tom Lieber, Kyle Murray

Lecture 8: Last Lecture Helter Skelter Fun!

January 31, 2012

Today...

- Standard Template Library (STL)
- Crazay Const
- Exceptions
- Function pointers
- Brief intro to C++11

Today...

- **Standard Template Library (STL)**
- Crazay Const
- Exceptions
- Function pointers
- Brief intro to C++11

Standard Template Library (STL)

- “Included” w/ compiler
- Contains containers/data structures, iterators, and algorithms
- <http://www.cplusplus.com/reference>

Vectors

- Equivalent to array lists in other languages (dynamic size)

```
#include <vector>
```

```
...
```

```
std::vector<int> int_list;
```

```
int_list.push_back(1);
```

```
int tmp = int_list[0]; // tmp = 1
```

```
int_list.pop_back(); // int_list now empty
```

Map

- Like a dictionary in Python (tree implementation).

```
#include <map>
```

...

```
std::map<char, int> letter_to_int;
```

```
letter_to_int['a'] = 1;
```

```
letter_to_int['b'] = 2;
```

```
int pos = letter_to_int['a'] // pos = 1;
```

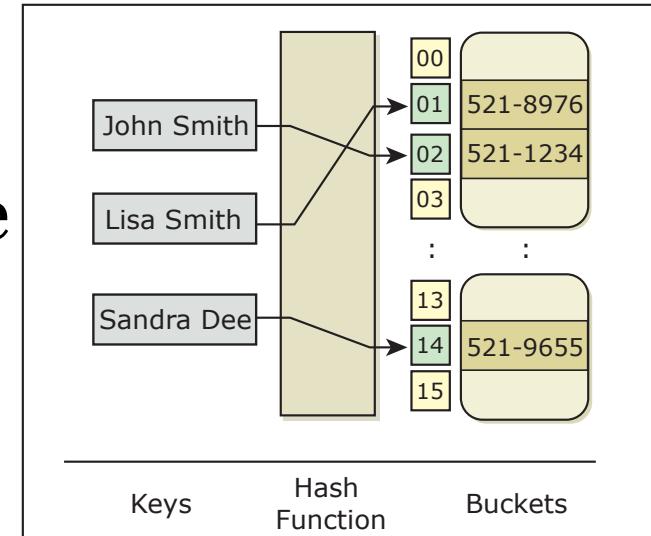


Image by MIT OpenCourseWare.

Others Containers

- Other useful containers:
 - **<array>**: array w/ functions
 - **<list>**: doubly linked list
 - **<set>**: stores only unique elements, tree implementation
 - **<unordered_map>**: actual hash table
 - **<unordered_set>**: stores only unique elements, hash table implementation
 - *Look online for more!*

Iterators

- Object that points to elements in a data structure, and can *iterate* through. Like a pointer.
- Vector iterator: `std::vector<int>::iterator it;`
- Access element at iterator: `*it;`
- Can do add/subtract to iterators: `it++, it--`

Vector Iterators

```
#include <vector>
#include <iostream>
...
std::vector<int> vec;
for (int i=1; i<=5; i++)
    vec.push_back(i);
for (std::vector<int>::iterator it = vec.begin();
     it != vec.end(); ++it){
    std::cout << ' ' << *it;
}
//Will print: 1 2 3 4 5
```

Cool/Useful Algorithm

- `#include <algorithm>`
- Sort a vector:
 - `std::sort(vec.begin(), vec.end());`
- Reverse a vector:
 - `std::reverse(vec.begin(), vec.end());`
- Min/Max:
 - `std::min(3,1) == 1 ; std::max(3,1) == 3`
- So many more online!

Today...

- Standard Template Library (STL)
- **Crazay Const**
- Exceptions
- Function pointers
- Brief intro to C++11

Const Madness!

- `const int x == int const x;`
`// Const int`
- `const int * c1;`
`// Pointer to const int`
- `int const * c2; // Same as c1`
- `int * const c3; // Const pointer to variable int`

More Const Madness!

- Const in function parameter

```
bool cant_touch_this(const myobj & obj);  
//cant_touch_this can't modify obj
```

- Const functions are safe for const objects, but can be called by all objects. Non-const functions can only be called by non-const objects.

```
bool catch_change_this() const {  
    ...  
}
```

When will the madness end!?!?

const int* const

crazay(const int* const & madness) const;

Today...

- Standard Template Library (STL)
- Crazay Const
- **Exceptions**
- Function pointers
- Brief intro to C++11

Exceptions

- Exceptions are “error” objects that are “thrown” when things go bad.

Exceptions

- Exception parent object in std::exception
 - std::runtime_error
 - std::bad_alloc
 - std::bad_cast
- Can create your own custom exceptions

```
class MyException : public exception{  
    const char * what() const {  
        return "MyException"; // human-readable  
    }  
};
```

Try Catch

```
try {  
    ... // Protected code  
    throw MyError();  
}catch( YourError e1 ){  
    cout << e1.what() << endl;  
}catch( MyError e2 ){  
    cout << e2.what() << endl;  
}catch(...) {  
    ... // handle all other exceptions  
}
```

Throwing that Exceptions

- Can throw a primitive as an exception

```
try{  
    ...  
    throw 20;  
}catch(int e) { cout << "Error: " << e << endl; }
```

- Best way to catch your own exception:

```
try{  
    throw MyException();  
}catch(MyException & e) {  
    cout << e.what() << endl;  
}
```

Functions Throwing

- Functions add `throw(exceptions-thrown)` at end of declaration.

```
void ahh(int i) throw() {  
    //This function assumed to not throw anything  
}
```

```
void blahh(int i) throw(int) {  
    //This function may throw int if there's an error  
}
```

Today...

- Standard Template Library (STL)
- Crazay Const
- Exceptions
- **Function pointers**
- Brief intro to C++11

Functions Pointers

- `void (*foo) (int);` // Foo is pointer to void function that takes 1 int argument.
- `void *(*foo)(int *, char);` //Any guesses?

Functions Pointers

- `void (*foo) (int);` // Foo is pointer to void function that takes 1 int argument.
- `void *(*foo)(int *, char);` //Foo is a pointer to a function that takes 1 int* argument and 1 char argument, that returns void*

Using Functions Pointers

```
#include <iostream>

void my_int_func(int x){
    std::cout << x << std::endl;
}

int main(){
    void (*foo)(int);
    foo = &my_int_func; // The ampersand is actually optional
    (*foo)(2); // Calls my_int_func
    foo(2); // Same as above line
    return 0;
}
```

Functions Pointers Arguments

```
#include <iostream>
void call(int x){
    std::cout << x << std::endl;
}
void call_me_maybe(int number, void (*call_func)(int)) {
    call_func(number);
}
int main(){
    void (*foo)(int);
    foo = call;
call_me_maybe(911, foo);
    return 0;
}
```

Functions Pointers Arguments

```
#include <iostream>

void call(int x){
    std::cout << x << std::endl;
}

void call_me_maybe(int number, void (*call_func)(int)) {
    call_func(number);
}

int main(){
    call_me_maybe(911, call);
    return 0;
}
```

For_each

```
#include <algorithm> // Header file need for for_each()
#include <vector>
#include <iostream>
using namespace std;
void print (int i) { cout << ' ' << i; }
int main(){
    vector<int> myvector;
    myvector.push_back(10);
    myvector.push_back(20);
    for_each (myvector.begin(), myvector.end(), print);
    ...
}
} // Prints out 10 20
```

Today...

- Standard Template Library (STL)
- Crazay Const
- Exceptions
- Function pointers
- **Brief intro to C++11**

C++11

- Latest and “greatest” standard of C++ w/ fancy features
- Fancy feature 1: Auto typing

```
int x = 1;  
auto y = x;  
vector<int> vec;  
auto itr = vec.begin(); //versus vector<int>::iterator
```

C++11

- Latest and “greatest” standard of C++ w/ fancy features
- Fancy feature 2: Declare typing

```
int x = 1;
```

```
decltype(x) y = x; //Makes y the same type as x
```

C++11

- Latest and “greatest” standard of C++ w/ fancy features
- Fancy feature 3: Right angle brackets
 - Before C++11:
`vector<vector<int> > vector_of_int_vectors;`
 - C++11:
`vector<vector<int>> vector_of_int_vectors;`

C++11

- Latest and “greatest” standard of C++ w/ fancy features
- Fancy feature 4: Range for loops

```
vector<int> vec;  
vec.push_back( 1 );  
vec.push_back( 2 );  
for (int& i : vec ) {  
    i++; // increments the value in the vector  
}
```

C++11

- Latest and “greatest” standard of C++ w/ fancy features
- Fancy feature 5: Lambda functions!!!
 - Syntax: [] (arg_list) { func_definition }
 - Ex:

```
#include <iostream>
using namespace std;
int main(){
    auto func = [] (int i) { cout << "Hi " << i << endl; };
    func(1); // now call the function
}
```

C++11

- Many more fancy features (strongly typed enums, null pointer constant, etc)!

\(^_^\) /

Yay!!!

- Reference:

<http://www.cprogramming.com/tutorial.html#c++11>

<http://en.wikipedia.org/wiki/C%2B%2B11>

It's about that time...

- Introduced you to C and C++ syntax, features, and idoms.
- Gave you some experience w/ writing C/C++ code
- Much more to learn, best way is to simply code more!

MIT OpenCourseWare
<http://ocw.mit.edu>

6.S096 Introduction to C and C++

IAP 2013

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.