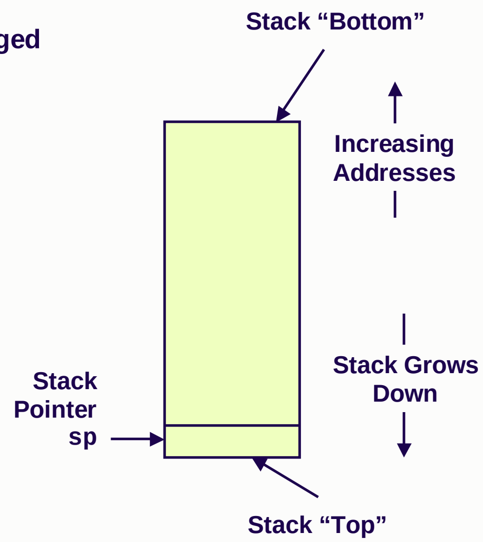
Function Calls and Stack Allocation

Topics

- Stack Pushing and Popping
- Role of Stack in Call Chain
- Stack (Automatic) Allocation
- Parameter Passing

Stack

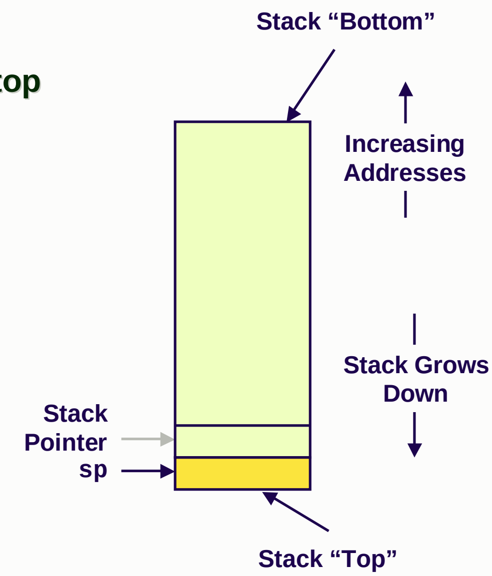
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Stack pointer indicates lowest stack address



Stack Pushing

Pushing

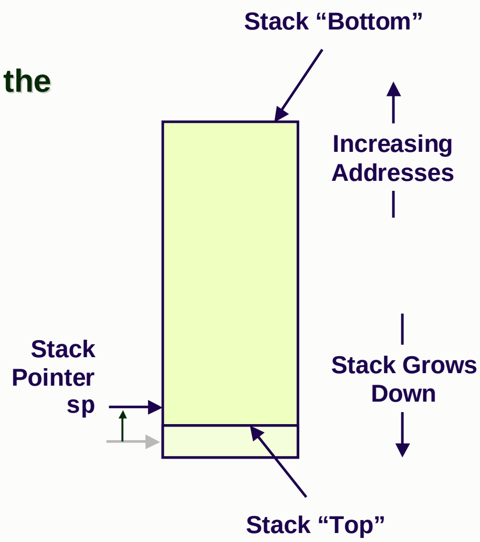
Add something at the top of the stack



Stack Popping

Popping

Throw away element at the top of the stack



Procedure Control Flow

- Use stack to support procedure call and return

Stack Allocated in *Frames*

- state for single procedure instantiation
  - Local variables
  - Arguments
  - Other (e.g., for return)
- all state goes away when procedure returns

Call Chain Example

Code Structure

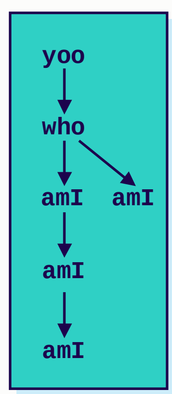
```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

```
who(…)
{
  • • •
  amI();
  • • •
  amI();
  • • •
}
```

```
amI(…)
{
  •
  •
  amI();
  •
  •
}
```

- Procedure `amI` recursive (calls itself)

Call Chain



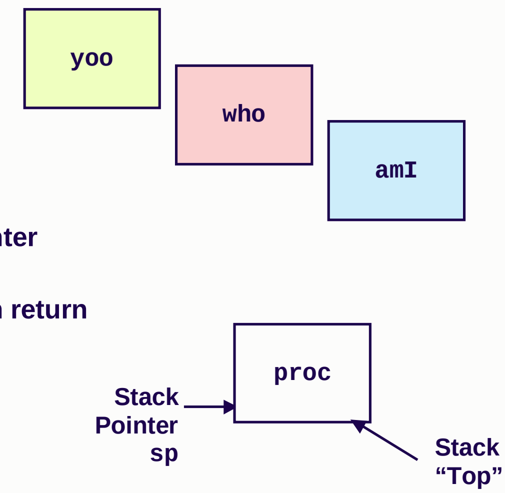
Stack Frames

Contents

- Local variables
- Return information
- Temporary space

Management

- Space allocated when enter procedure
- Deallocated (freed) when return

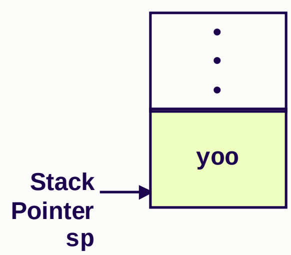


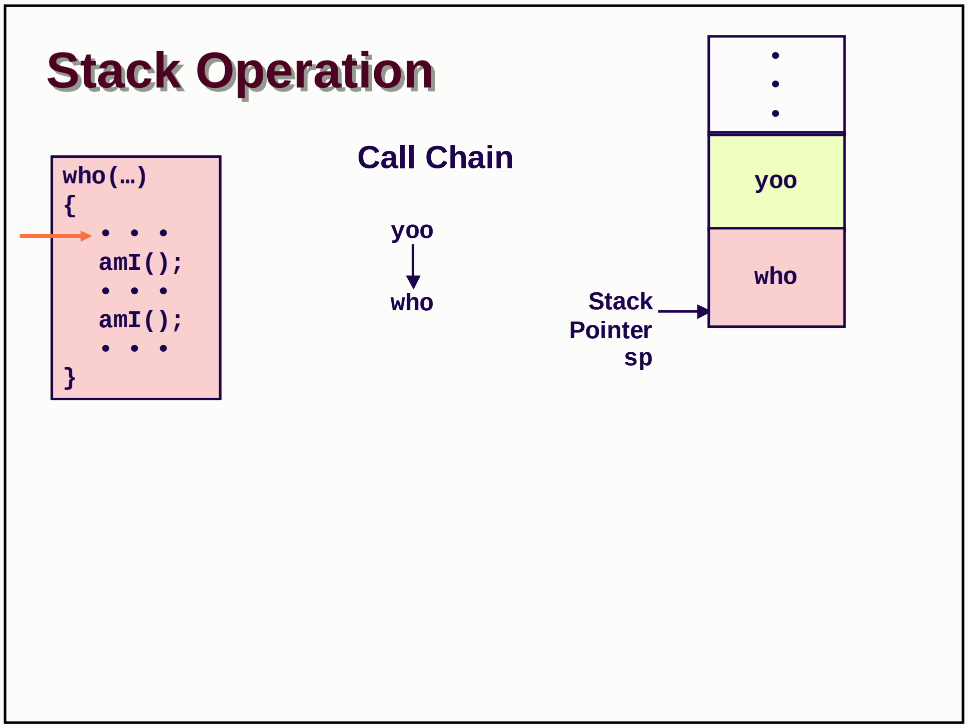
Stack Operation

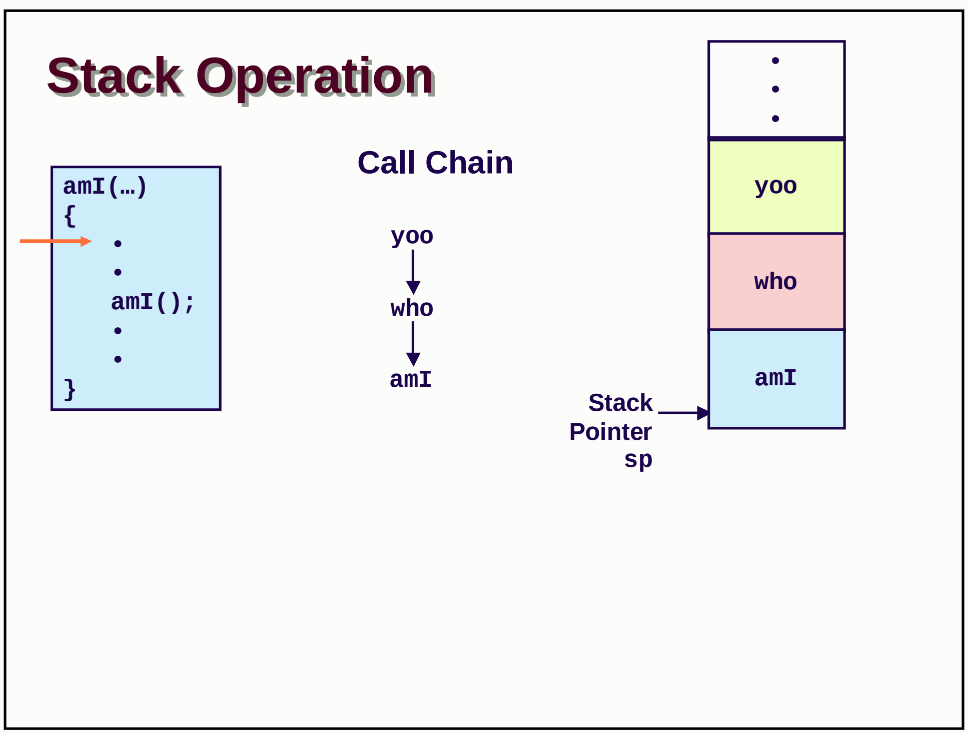
```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

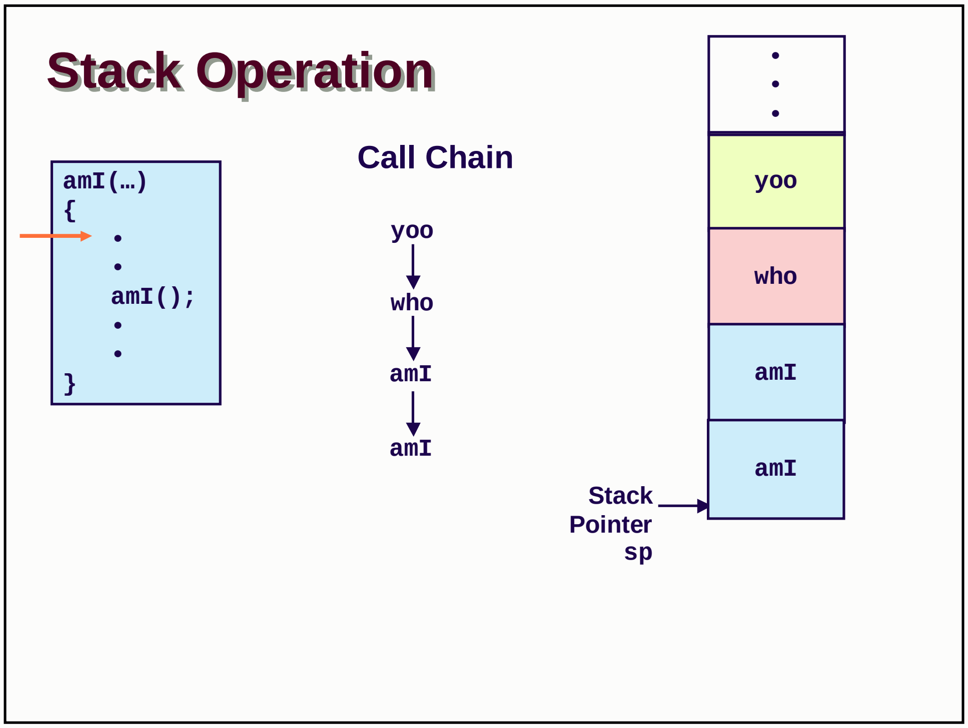
Call Chain

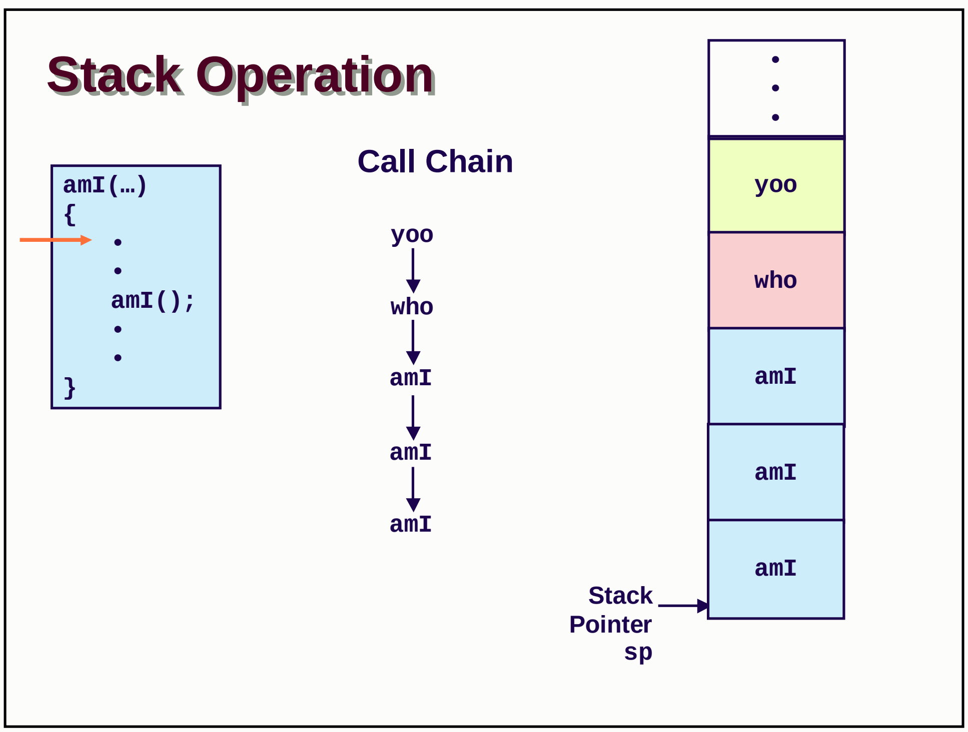
yoo

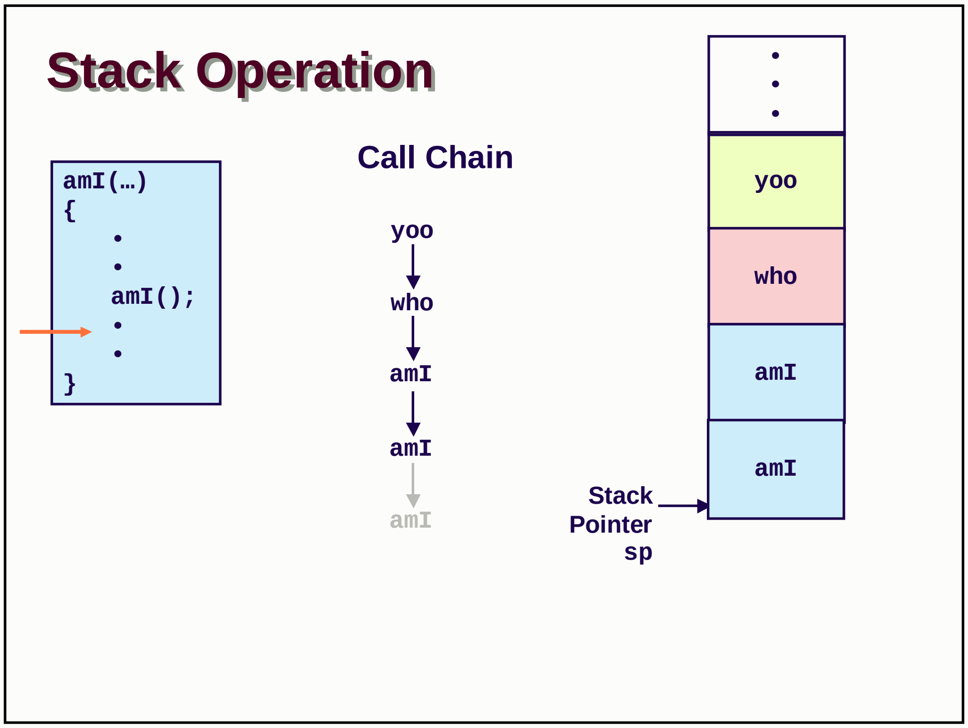


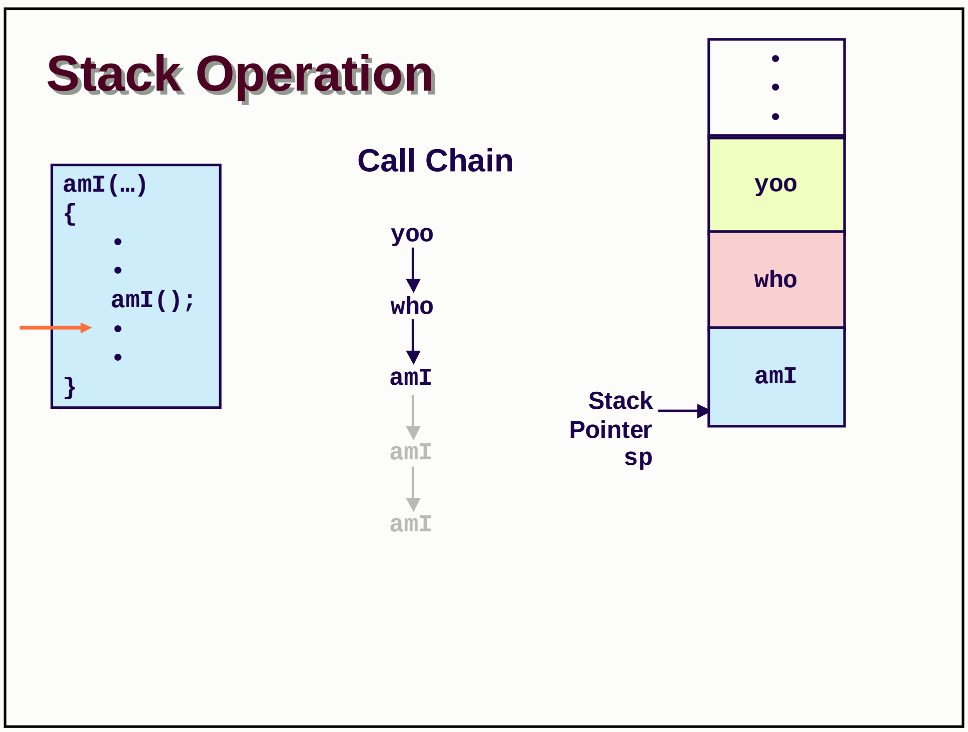

Stack Operation
who(…)
{
• • •
amI();
• • •
amI();
• • •
}
Call Chain
yoo
who
•
•
•
yoo
who
Stack
Pointer
sp


Stack Operation
amI(…)
{
•
•
amI();
•
•
}
Call Chain
yoo
who
amI
•
•
•
yoo
who
amI
Stack
Pointer
sp


Stack Operation
amI(…)
{
•
•
amI();
•
•
}
Call Chain
yoo
who
amI
amI
•
•
•
yoo
who
amI
amI
Stack
Pointer
sp

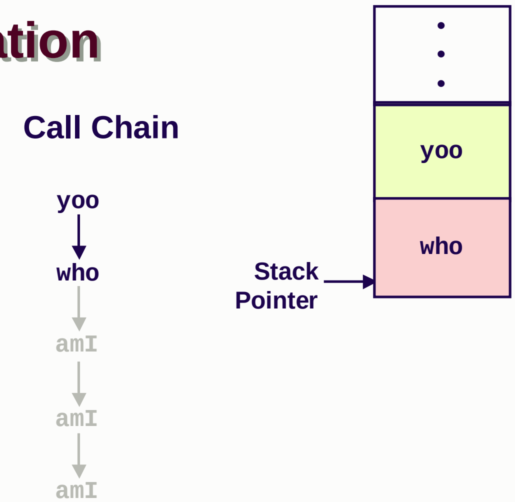

Stack Operation
amI(…)
{
•
•
amI();
•
•
}
Call Chain
yoo
who
amI
amI
amI
•
•
•
yoo
who
amI
amI
amI
Stack
Pointer
sp


Stack Operation
amI(…)
{
•
•
amI();
•
•
}
Call Chain
yoo
who
amI
amI
amI
•
•
•
yoo
who
amI
amI
Stack
Pointer
sp


Stack Operation
amI(…)
{
•
•
amI();
•
•
}
Call Chain
yoo
who
amI
amI
amI
•
•
•
yoo
who
amI
Stack
Pointer
sp

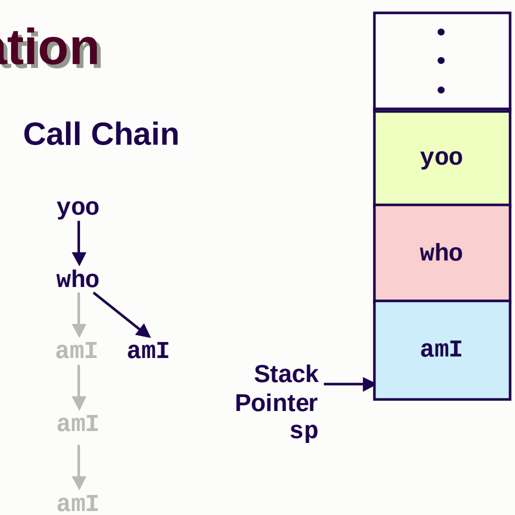
Stack Operation

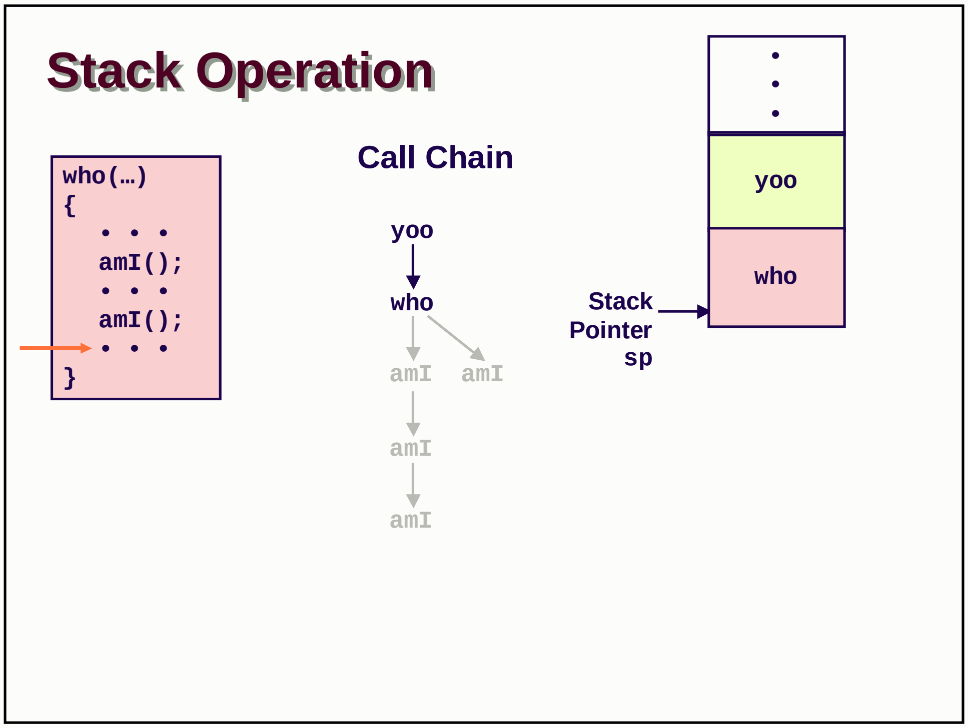
```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

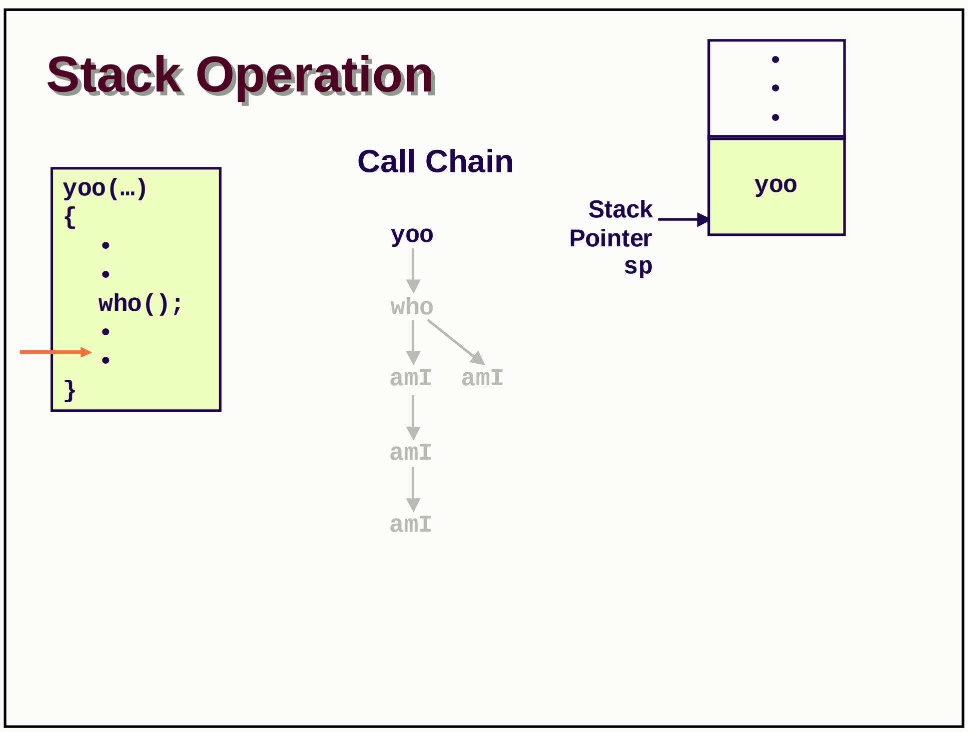


Stack Operation

```
amI(…)
{
    •
    •
    •
    •
}
```




Stack Operation
who(…)
{
• • •
amI();
• • •
amI();
• • •
}
Call Chain
yoo
who
amI
amI
amI
amI
Stack
Pointer
sp
yoo
who


Stack Operation
yoo(…)
{
•
•
who();
•
•
}
Call Chain
yoo
who
amI
amI
amI
amI
Stack
Pointer
sp
yoo

Function Parameters

Function arguments are passed "by value".

What is "pass by value"?

- **The called function is given a copy of the arguments.**

What does this imply?

- **The called function can't alter a variable in the caller function, but its private copy.**

An example

Example 1: swap_1

```
void swap_1(int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

Q: Let x=3, y=4,
after swap_1(x,y);
x =? y=?

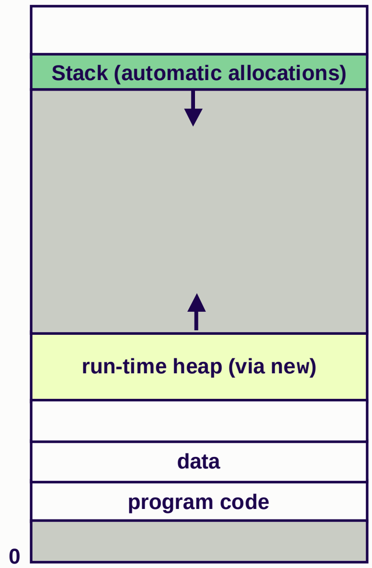
~~A1: x=4; y=3;~~

A2: x=3; y=4;

Process Memory Image

Automatic variables are allocated memory on the stack.
Stack grows downwards

Dynamic Memory Allocator requests memory from the heap.
Heap grows upwards



Dynamic (Heap) Memory Allocator Recap

Operator new is still a high-level request such as "I'd like an instance of class `String`"

Try to think about it low level

- You ask for *n* bytes (the sizeof that type/class)
- You get a pointer (memory address) to the allocated object
- This allocation is on the heap
- You need to free all memory blocks you allocated
  - A delete for each corresponding new

Automatic Allocator Internals

Automatic allocation of variables occurs on the stack

We'll learn how automatic allocation works next

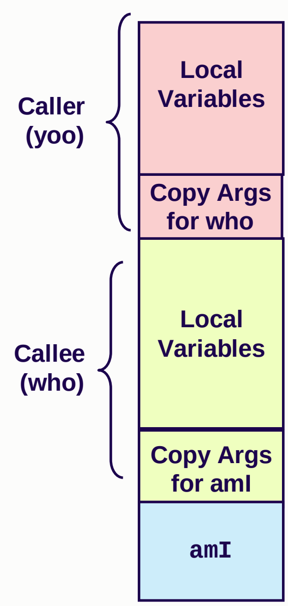
Passing Arguments by Value

Upon function call, the argument values are copied (byte-by-byte) onto stack

- Push args on stack for the function you are about to call

Space for local variables is allocated on stack

- Local variables allocated automatically in new frame
- Disappear when frame pops off the stack



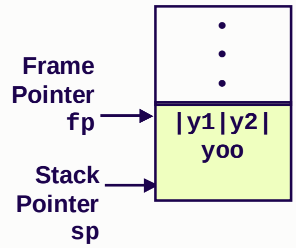
Pass Args by Value

```
yoo()
{ int y1,y2;
   •
   •
   who(y1,y2);
   •
   •
}
```

```
who(int a, int b)
{
 int w1,w2;
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

Call Chain

yoo



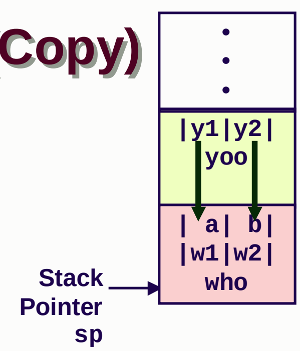
Pass Args by Value (Copy)

```
yoo()
{ int y1,y2;
   •
   •
   who(y1,y2);
   •
   •
}
```

```
who(int a, int b)
{
 int w1,w2;
   • • •
   amI(a);
   • • •
   amI(b);
   • • •
}
```

Call Chain

yoo → who



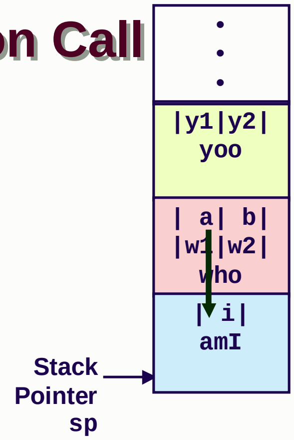
Pass Args by Value on Call

```
who(int a, int b)
{
 int w1,w2;
   • • •
   amI(a);
   • • •
   amI(b);
   • • •
}
```

```
amI(int i)
{
    •
    •
    amI(i);
    •
    •
}
```

Call Chain

yoo → who → amI

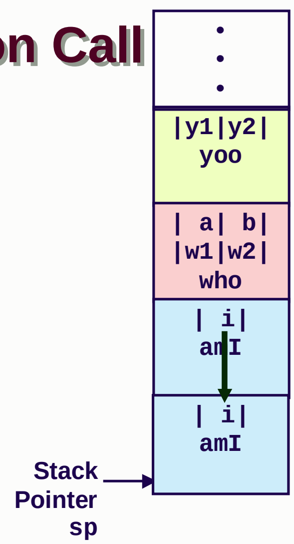


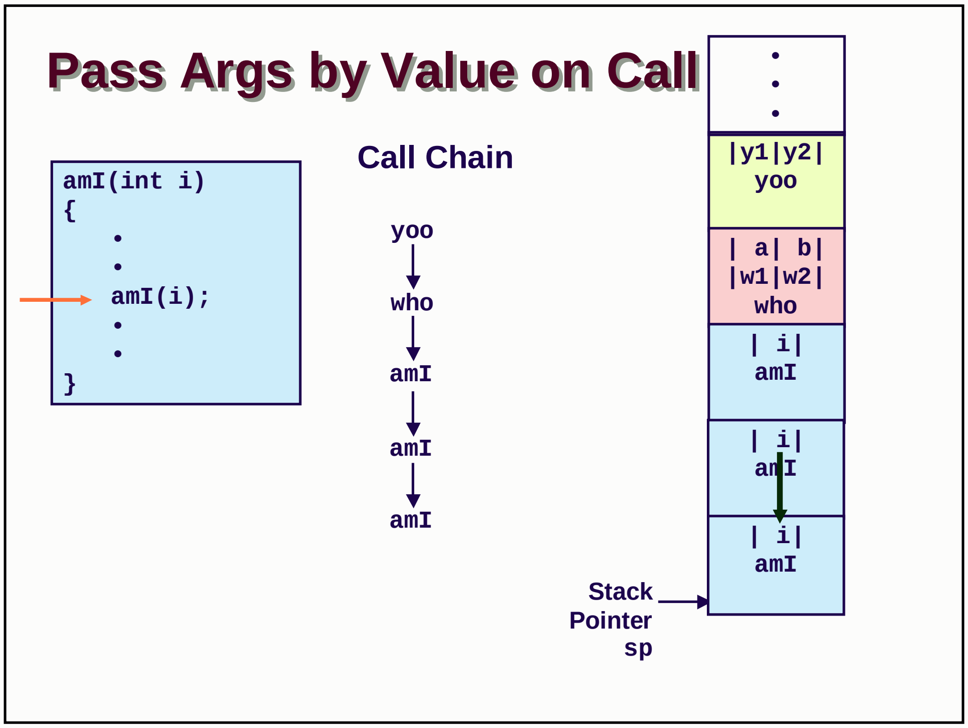
Pass Args by Value on Call

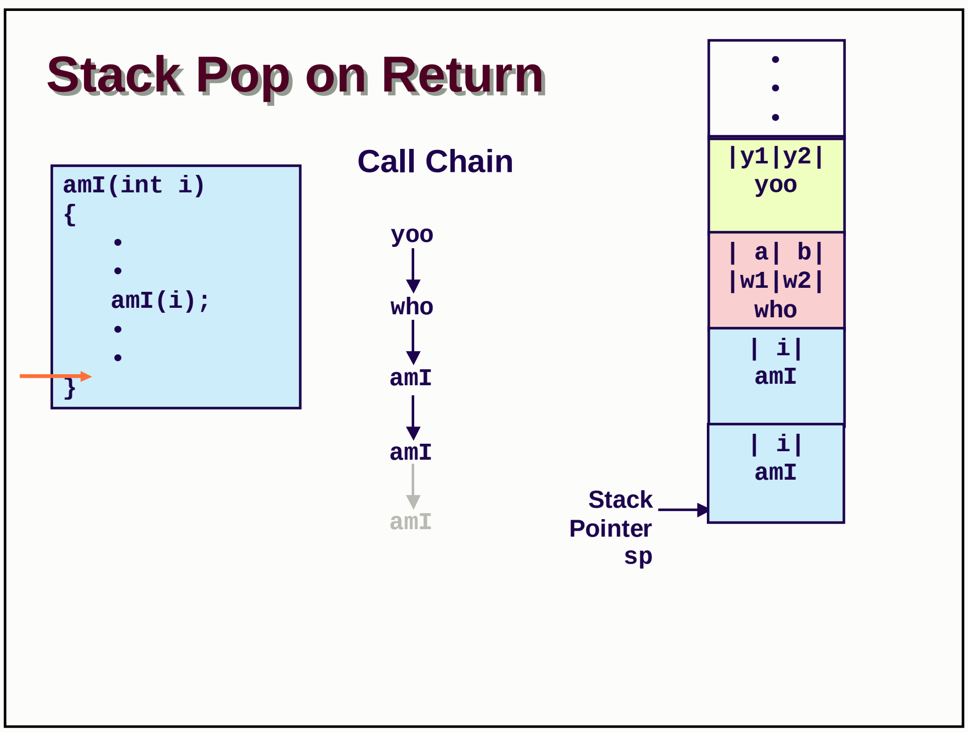
```
amI(int i)
{
    •
    •
    amI(i);
    •
    •
}
```

Call Chain

yoo → who → amI → amI

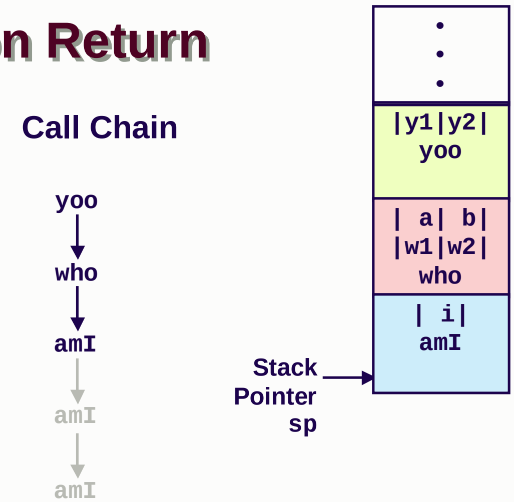



Pass Args by Value on Call
amI(int i)
{
•
•
amI(i);
•
•
}
Call Chain
yoo
who
amI
amI
amI
•
•
•
|y1|y2|
yoo
| a| b|
|w1|w2|
who
| i|
amI
| i|
amI
| i|
amI
Stack
Pointer
sp


Stack Pop on Return
amI(int i)
{
•
•
amI(i);
•
•
}
Call Chain
yoo
who
amI
amI
amI
•
•
•
|y1|y2|
yoo
| a| b|
|w1|w2|
who
| i|
amI
| i|
amI
Stack
Pointer
sp

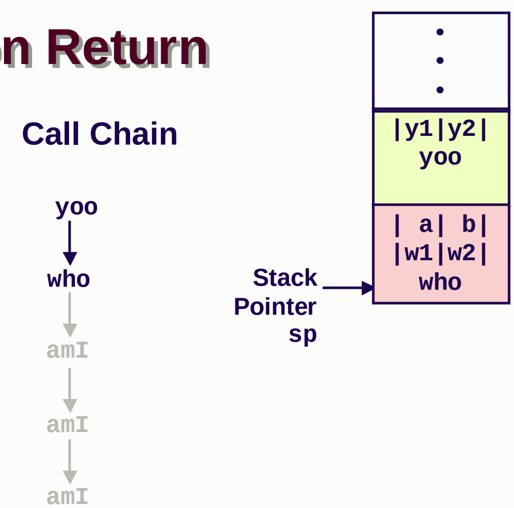
Stack Pop on Return

```
amI(int i)
{
    •
    •
    amI(i);
    •
    •
}
```



Stack Pop on Return

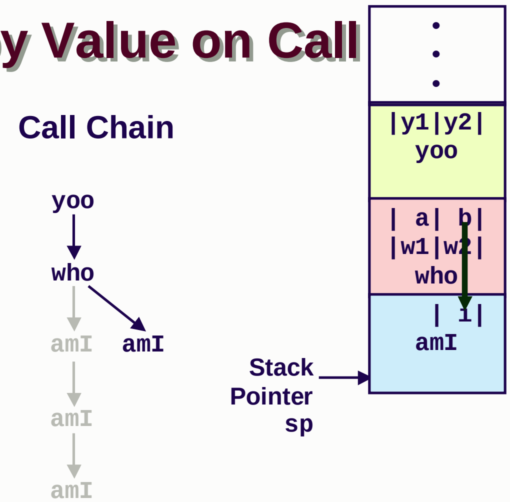
```
who(int a, int b)
{
 int w1,w2;
   • • •
   amI(a);
   • • •
   amI(b);
   • • •
}
```



Pass Args by Value on Call

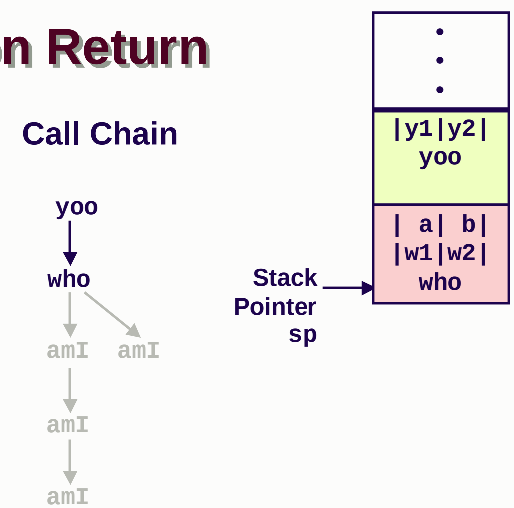
```
who(int a, int b)
{
 int w1,w2;
   • • •
   amI(a);
   • • •
   amI(b);
   • • •
}
```

```
amI(int i)
{
    •
    •
    amI(i);
    •
    •
}
```



Stack Pop on Return

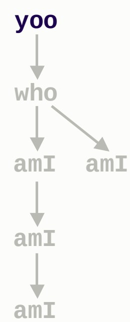
```
who(int a, int b)
{
 int w1,w2;
   • • •
   amI(a);
   • • •
   amI(b);
   • • •
}
```

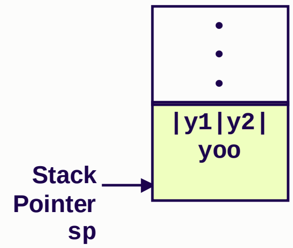


Stack Pop on Return

```
yoo()
{ int y1,y2;
   •
   •
   who(y1,y2);
   •
   •
}
```

Call Chain





Summary

The Stack Makes Function Calls Work

- Private storage for each *instance* of procedure call
  - locals + arguments are allocated on stack
- Can be managed by stack discipline
  - Procedures return in inverse order of calls
- That’s how automatic allocation works
  - Local vars allocated on new frame upon entering function call
  - Vars (including arg copies) freed automatically upon return

Do you see now why you cannot delete an automatically allocated object ?

Summary (contd.)

Do you see now why you cannot delete an automatically allocated object ?

(e.g. int i; int * pi = &i; delete pi is WRONG !)

Because automatically allocated objects live temporarily on the stack. You cannot control lifetime.

You can only free objects that you allocated with new (on the heap).

The two allocators (dynamic & automatic) are different.

Function Parameters Passing (contd)

The only mechanism in C++ is to pass arguments by value (push/copy args on stack) !!!

So how can we make swap work ?

- A: The called function is passed a pointer (address) of a var.

What does this imply?

- The called function can alter that variable var through its pointer
- This fakes a mechanism called "pass args by reference" present in other languages (e.g., Pascal).

An example

Example 2: swap_2

```
void swap_2(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Q: Let x=3, y=4,
after
swap_2(&x,&y);
x =? y=?

~~A1: x=3; y=4;~~

A2: x=4; y=3;

Parameters Passing "by Reference"

1. **The stack mechanism works unchanged**
2. **The pointer (arg) is still copied (byte by byte) on stack as usual !**
3. **So the pointer itself is still passed "by value"**
4. **However, the callee can directly access that memory address**
   - **thus can change the var through its pointer**
5. **If arg is large object (e.g., struct student_data) should pass its address (to avoid large copies)**

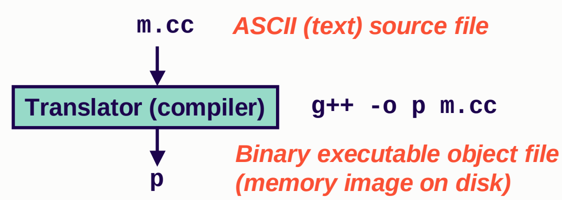
Compilation and Linking

Topics

Separation of function code into .cc and .h

- Compiling
- Object files
- Linking different files

A Simplistic Program Translation Scheme (seen up to now)



Problems:

- Efficiency: small change requires complete recompilation
- Modularity: hard to share common functions (e.g., cout, sort)

Solution:

- *Use separate files for different functionalities (code is "modular")*
- *Linker*

Example C Program

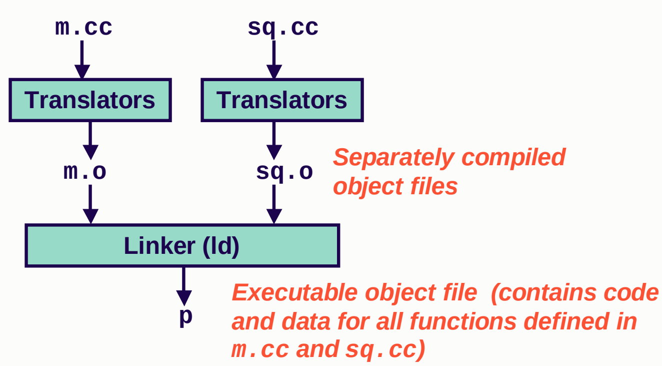
m.cc

```
int e=7;

int main() {
  ...
 z = squared(y);
 cout << e;

 return 0;
}
```

sq.cc

```
extern int e;

int squared(int y) {
  return y*y;
}
```

A Better Scheme Using a Linker



Translating the Example Program

g++ coordinates all steps in the translation and linking process.

- Invokes preprocessor, compiler, assembler (as), and linker (`ld`).
- Passes command line arguments to appropriate phases

Example: create executable p from `m.cc` and `sq.cc`:

➢g++ -c m.cc	This creates m.o
➢g++ -c sq.cc	This creates sq.o
➢g++ m.o sq.o –o p	Links m.o and sq.o

Translating the Example Program

g++ coordinates all steps in the translation and linking process.

- Invokes preprocessor, compiler, assembler (as), and linker (`ld`).
- Passes command line arguments to appropriate phases

Example: create executable p from `m.cc` and `sq.cc`:

➢g++ -c m.cc	This creates m.o
➢g++ -c sq.cc	This creates sq.o
➢g++ m.o sq.o –o p	Links m.o and sq.o

```
Can do it like this too:
➢g++ m.cc sq.cc –o p
But:
If one file changes,all need to be recompiled,long compile time
```

What Does a Linker Do?

Merges object files

- Merges multiple (.o) object files into a single executable object file that can be loaded and executed.

Resolves external references

- As part of the merging process, resolves external references.
  - *External reference*: reference to a symbol defined in another object file.
  - External references can be to either code or data
    - » code: `a();          /* reference to symbol a */`
    - » data: extern `int x;   /* reference to symbol x */`

Why Linkers?

Modularity

- Program can be written as a collection of smaller source files, rather than one monolithic mass.
- Can build libraries of common functions (more on this later)
  - e.g., Math library, iostream library

Efficiency

- Time:
  - Change one source file, recompile that one !, and then relink.
  - No need to recompile other source files
    - » e.g., if Bahlul changes sort, then only his file will be recompiled to produce his .o, not our own

So what goes in .cc and in .h ?

main.cc

```
int main() {
Listnode *head;
  ...
 z = squared(y);

 head = free_list(head);

 return 0;
}
```

squared.cc

```
int squared(int y) {
  return y*y;
}
```

list.cc

```
typedef struct list_node {
. . .
} Listnode;

Listnode *free_list(Listnode *l){
 ..//ex5 use no aux pointers
  //ex4&5 prize Visual C++ free
}
```

Example C++ Program

main.cc

```
int main() {
Listnode *head;//Listnode ??
  ...
 z = squared(y);

 head = free_list(head);

 return 0;
}
```

squared.cc

```
int squared(int y) {
  return y*y;
}
```

list.cc

```
typedef struct list_node {
. . .
} Listnode; //move to list.h

Listnode *free_list(Listnode *l){
 .. //ex5 use no aux pointers
}
```

Example Program

main.cc

```
#include "squared.h"

int e=7;

int main() {
Listnode *head;
  ...
 z = squared(y);

 head = free_list(head);

 return 0;
}
```

squared.h

```
int squared(int y);
```

squared.cc

```
#include "squared.h"

int squared(int y) {
  return y*y;
}
```

Example Program

main.cc

```
#include "squared.h"
#include "List.h"
int e=7;

int main() {
Listnode *head;
  ...
 z = squared(y);

 head = free_list(head);

 return 0;
}
```

List.h

```
typedef struct list_node {
. . .
} Listnode;

Listnode *free_list(Listnode *l);
```

list.cc

```
#include "List.h"

Listnode *free_list(Listnode *l){
 .. //ex5 use no aux pointers
}
```

How It Works

main.cc

```
#include "List.h"
#include "squared.h"
int e=7;

int main() {
Listnode *head;
  ...
 z = squared(y);

 head = free_list(head);

 return 0;
}
```

List.h

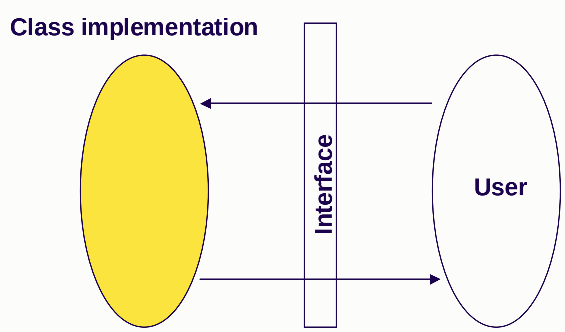
```
typedef struct list_node {
. . .
} Listnode;

Listnode *free_list(Listnode *l);
```

squared.h

```
int squared(int y);
```

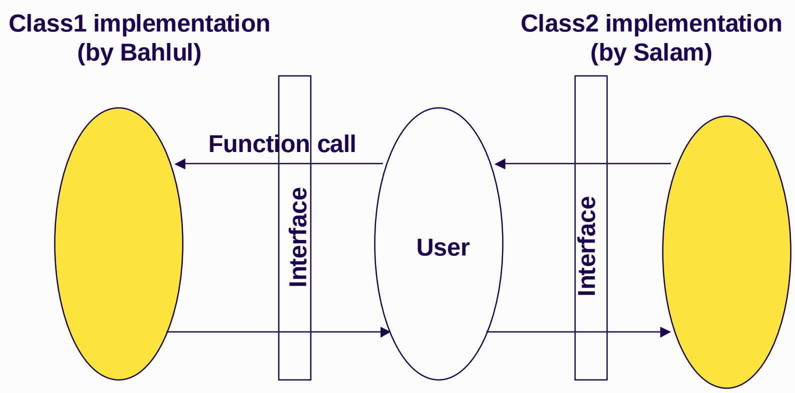
Preprocessor includes all text in List.h and squared.h in main.cc. We just separate declarations out for use by others and for the benefit of compiler/linker.

Encapsulation Introduction



Implementation details hidden from User

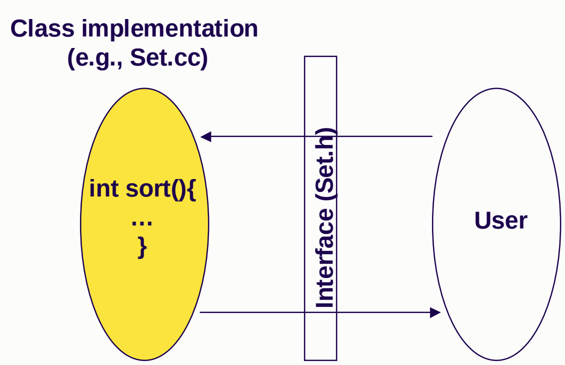
Encapsulation Introduction



Implementation details hidden from User

Programmers collaborate (use each other's code thru func calls)

Encapsulation



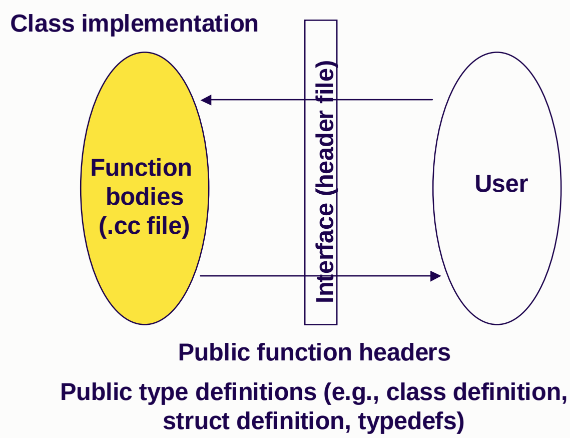
Implementation details (in Set.cc) hidden from User

Interface (in Set.h) is public - given to User

How to Encapsulate (first step) ?

Interface = spec (how to use) put in a header file

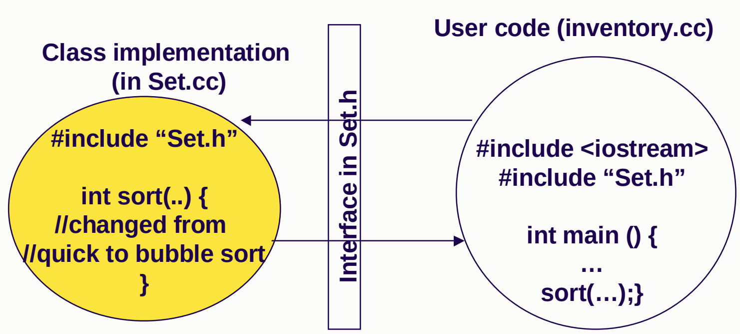
We'll see more of this with classes (e.g., private/public)



+/- of Encapsulation ?

+ User code remains same if class implem changes

- User code might run slower without apparent reason



Bottom Line: Code is spread out over several .cc and .h files
Changes to implementation are transparent to user
Only the file that changes needs to be recompiled