Slides by Andrew DeOrio,
Jeff Ringenberg and
Brian Noble

EECS 280

Programming and Introductory Data Structures

Memory Models

Static, fixed-size structures

- So far, the data structures we've built have all had room for "at most N" elements:
- The various `IntSet` implementations could have at most 100 distinct integers

- We could extend these sizes to larger ones, but we really only know how to create *static, fixed-sized* structures.

Static, fixed-size structures

- Sometimes, the process you are modeling has a physical limit, which makes a static, fixed-sized structure a reasonable choice.
- For example, a deck of cards has 52 individual cards in it, so this is a reasonable limitation.

- However, there is no meaningful sense in which a "set of integers" is limited to some particular number of elements.
- No matter how big you make the set's capacity, an application that needs more will eventually come along.
- Consider the `list_t` type from the second project:
  - The type imposed no limits on how large a list could grow

Global and local variables

- For the variables we have used so far you need to know two things at compile time (statically)
  - The size (or number)
  - The lifetime (when it will be created and destroyed)

- There have been two classes of such variables:
  1. Global variables
  2. Local variables

Global variables

- Global variables are defined anywhere outside of a function definition
- Space is set aside for these variables before the program begins execution, and is reserved for them until the program completes
- This space is reserved at compile time

Global variable examples

```
const int SIZE=10;
int main() {
  //...
}
```

```
static Triangle
  g_triangle;
Shape * ask_user() {
  //...
  return &g_triangle;
}
```

```
int sum_cur=0; //bad idea
int sum(list_t list){
  // ...
}

int prod_cur=0;//bad idea
int product(list_t list){
  // ...
}
```

Local variables

- Local variables are any variable defined within a block
- This includes function arguments, which act as if they were defined in the outermost block of the function
- Space is set aside for these variables when the relevant block is entered, and is reserved for them until the block is exited
- This space is reserved at run time, but the size is known to the compiler

- Since the compiler must know how big all of these variables will be, it is static information, and must be declared by the programmer

Local variable examples

```
int sum(list_t list){
  //...
}

int sum(int *array, int size) {
  int sum=0;
  //...
}

int main(int argc, char* argv[]) {
  IntSet i;
  for (int i=0; i<10; ++i) {
    //...
  }
}
```

Dynamic variables

- There is a third type of variable, called a *dynamic variable*
- It is dynamic because
  - Size (or number) is determined at runtime
  - When it will be created and destroyed is determined at runtime

- In other words, you (the programmer) get to decide how big a dynamic variable is, when it is created, and when it is destroyed

new

- Create dynamic variables using `new`

```
int main() {
  int *p = new int;
}
```

- This creates new space for an integer, and returns a pointer to that space, assigning it to `p`
- The initial value is undefined

- Use initializer syntax to assign an initial value

```
int main() {
  int *p = new int(5);
}
```

Exercise

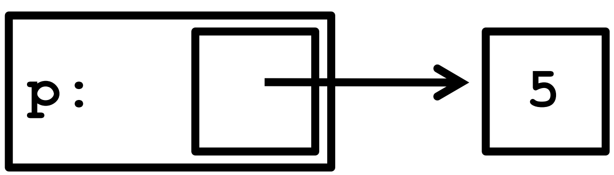
```
int main() {
  int *p = new int(5);
}
```

- How many variables are in this program?
- Mark each as global, local or dynamic

Exercise

```
int main() {
  int *p = new int(5);
}
```

- How many variables are in this program? **2**
  1. `int *p` is a local variable of type "pointer to integer"
  2. `int(5)` is a dynamic variable of type "integer"
- This “thing pointed to by `p`” is a dynamically-allocated piece of memory, and lives “somewhere else”.
- It does not have a name of its own, but is “the integer pointed to by `p`”.



delete

- Destroy dynamic variables using `delete`

```
int *p = new int;
//do something with p
delete p;
```

- Releases the claim on the space previously used by the `int`
- Space can be recycled later by `new`

delete pitfalls

- You can only `delete` a dynamic variable *once*

```
int *p = new int;
//do something with p
delete p;
delete p; //Error!
```

```
*** glibc detected ***
./a.out: double free or
corruption (fasttop):
0x00000000007ac010 ***
...
Aborted (core dumped)
```

delete pitfalls

- `delete 0` (AKA `NULL`) does nothing

```
int *p = new int;
//do something with p
delete p; p=0;
delete p; //OK
```

delete pitfalls

- Ordinary objects can be destroyed by `delete`, *but only if they were created by* `new`

```
int i = 0; //local variable
int *p=&i; //pointer to local variable
delete p;  //undefined! (likely a runtime error)
```

Size of dynamic variables

- With dynamic arrays, we can choose the size at runtime

```
//ask user to enter integer size
int size = get_size_from_user();

int *p = new int[size];
//do something with p ...
delete[] p;
```

- Note the different syntax with `delete`
- We will talk about dynamic arrays in more depth next time

Lifetime of dynamic variables

- What's the problem?

```
//EFFECTS: allocates an array of specified size
//  and initializes each element to zero
int * get_zero_array(int size) {
  int array[size];
  for (int i=0; i<size; ++i) array[i] = 0;
  return array;
}
int main() {
  int *a = get_zero_array(3);
  cout << a[0] << endl;
}
```

Lifetime of dynamic variables

- The lifetime of an object is completely under the control of the programmer – it lives until it is explicitly destroyed
- You can create a variable in one function and use it in another

```
//EFFECTS: allocates an array of specified size
//  and initializes each element to zero
int * get_zero_array(int size) {
  int *array = new int[size];
  for (int i=0; i<size; ++i) array[i] = 0;
  return array;
}
```

Lifetime of dynamic variables

```
//EFFECTS: allocates an array of specified size
//  and initializes each element to zero
int * get_zero_array(int size) {
  int *array = new int[size];
  for (int i=0; i<size; ++i) array[i] = 0;
  return array;
}

int main() {
  int *a = get_zero_array(10);
  //use a
  delete[] a; a=0;
}
```

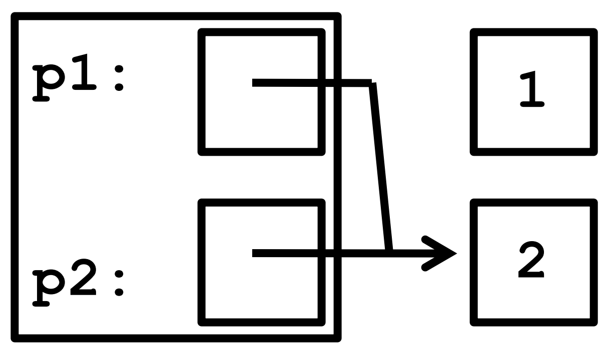
Call `new` in one function, and call `delete` in another function

Memory leaks

- Dynamic variables live until the programmer destroys them using `delete`
- This is true even if you "forget" the pointer to the object.

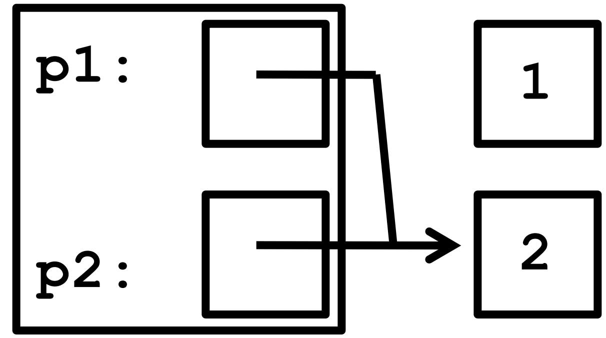
```
int main() {
  int *p1 = new int(1);
  int *p2 = new int(2);
  p1 = p2;
}
```

- This leaves us with:



Memory leaks

```
int main() {
  int *p1 = new int(1);
  int *p2 = new int(2);
  p1 = p2;
}
```

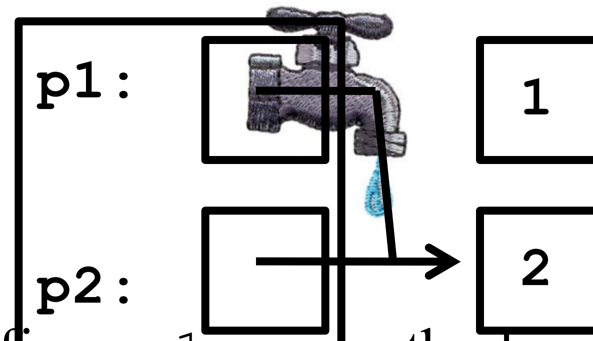


- Two pointers point to the object "2", and **none** to the object "1"
- There is no way to release the memory occupied by "1"

Memory leaks

- Note there is an important difference between the lifetime of a pointer variable and the lifetime of the object it points to!

```
int main() {
  int *p1 = new int(1);
  int *p2 = new int(2);
  p1 = p2;
}
```



- In this example, exiting the block that defines `p1` causes the local object `p1` to vanish, but the dynamic object it points to remains
- This leaves us with an allocated dynamic object that we have no means of reclaiming called a memory leak
- If memory leaks occur often enough, your program may reach a point where it can no longer allocate new dynamic objects

Why use dynamic variables?

1. If you want to manage the lifetime of the variable yourself
   - Not stack, Not global, but something in between

2. Many parts of the code refer to an object, but only want one copy in memory
   - Normally, if many parts refer to the same object, those parts will be in different scopes

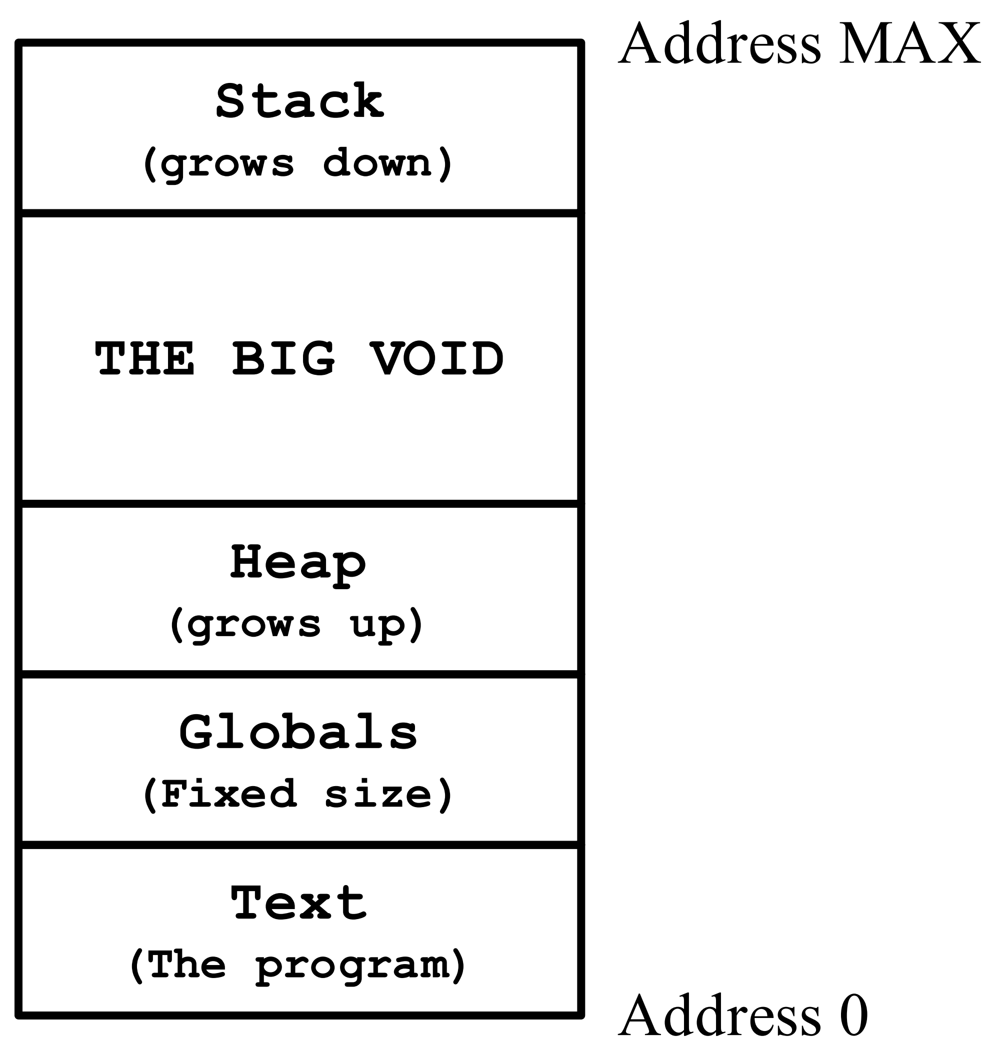
The heap

- The space for objects created via `new` comes from a location in memory called the heap.

- To describe the heap, we first have to revisit the memory model used by a "typical" C++ process:
- A running program has an "address space", a collection of memory locations that are accessible to it
- An address space is private to a running program – no other running program can access/modify it

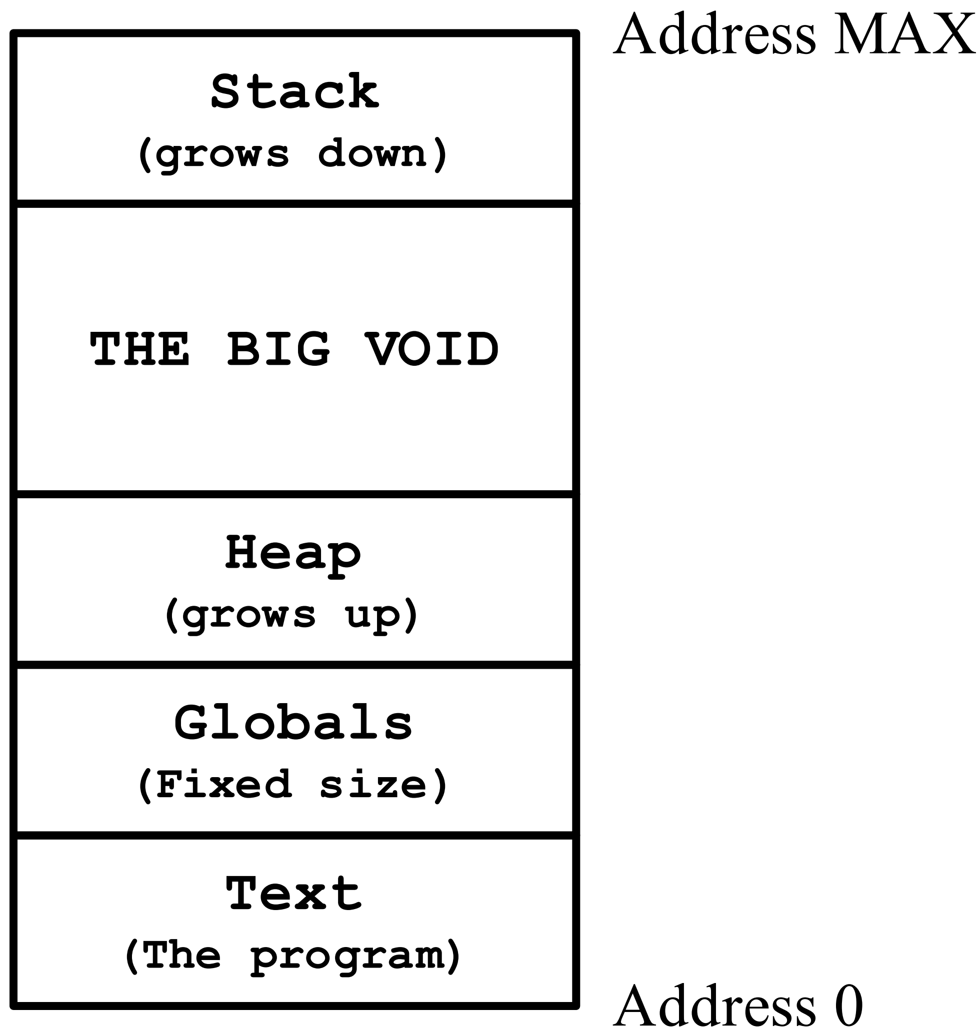
Program segments

- There are typically five parts, called segments, in an address space:



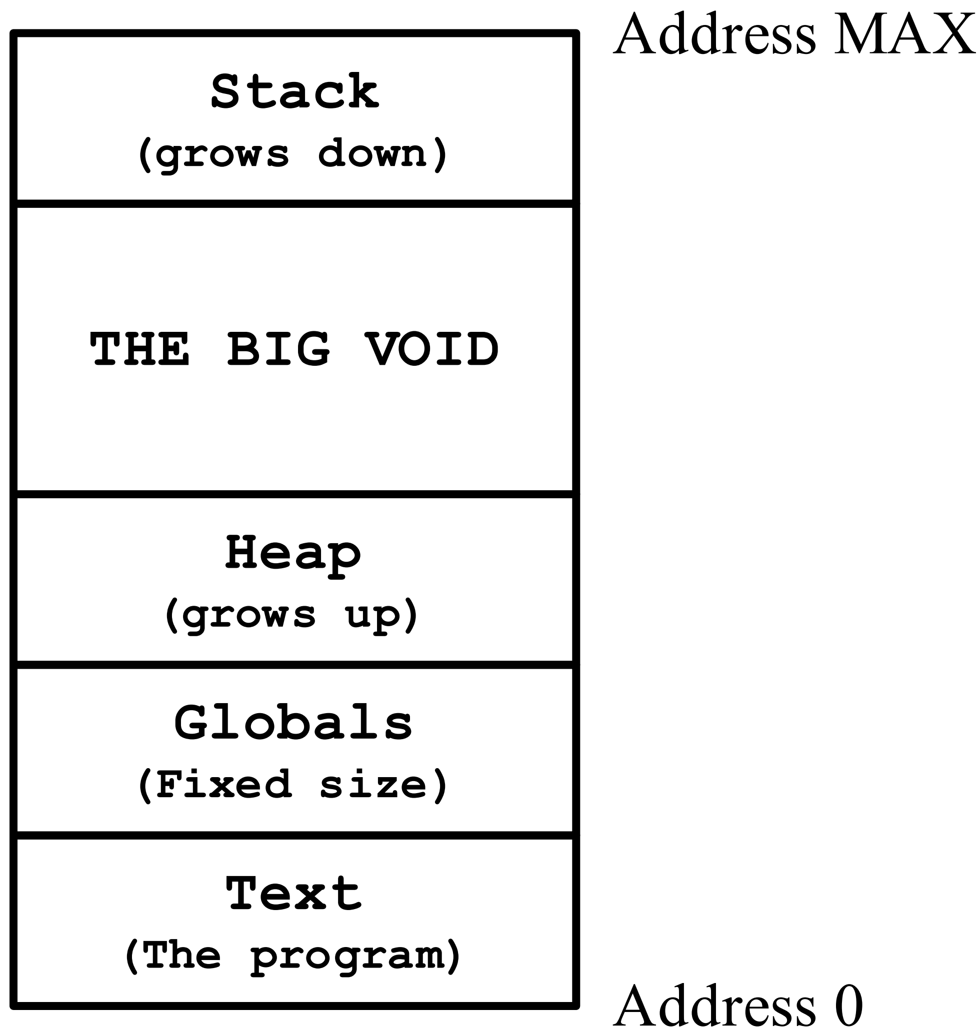
Text segment

- The code comprising a compiled program goes in the text segment.
- It is at a very low address, but typically not at address zero.



Globals and heap segments

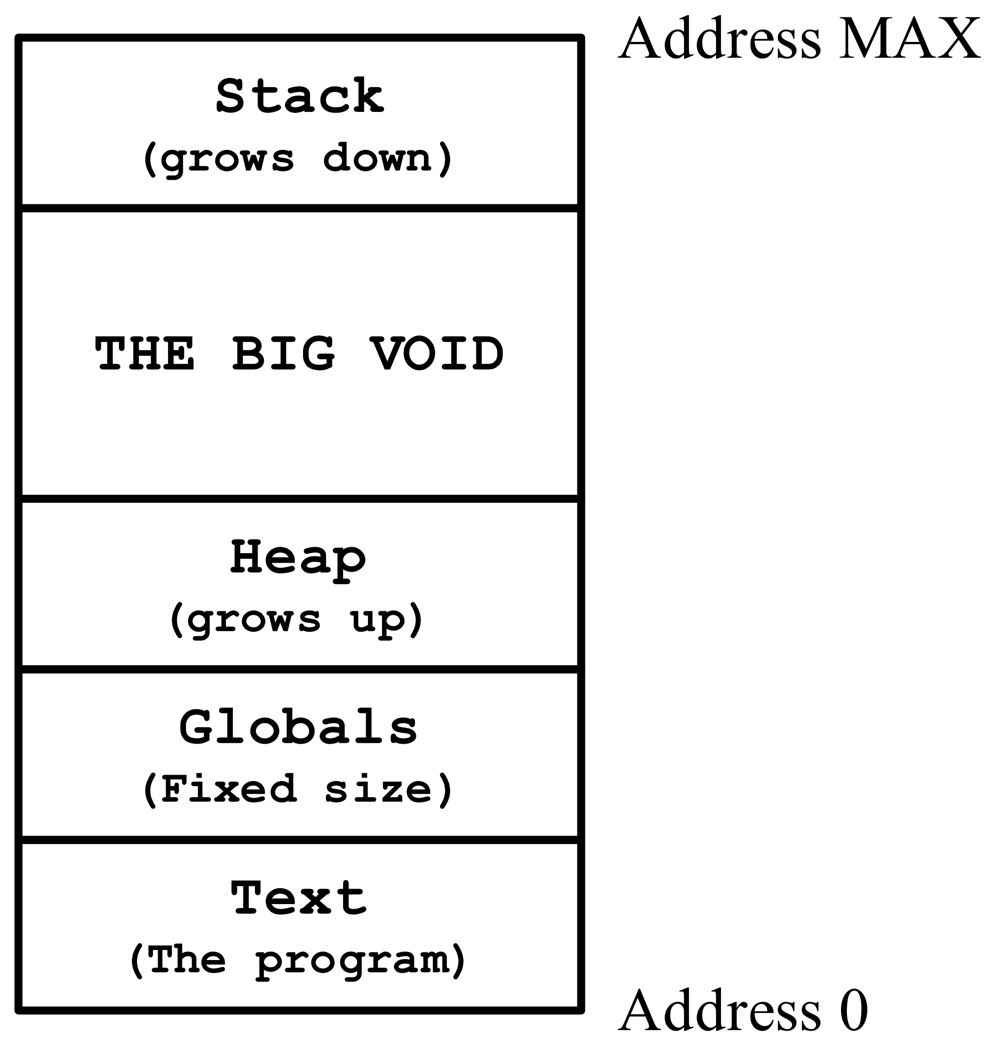
- Immediately above the *text* section, the compiler allocates space for any global variables, and initializes them, when necessary.
- When dynamic variables are allocated with `new`, they come from the *heap*, which grows upward.



Stack segment and the Big Void

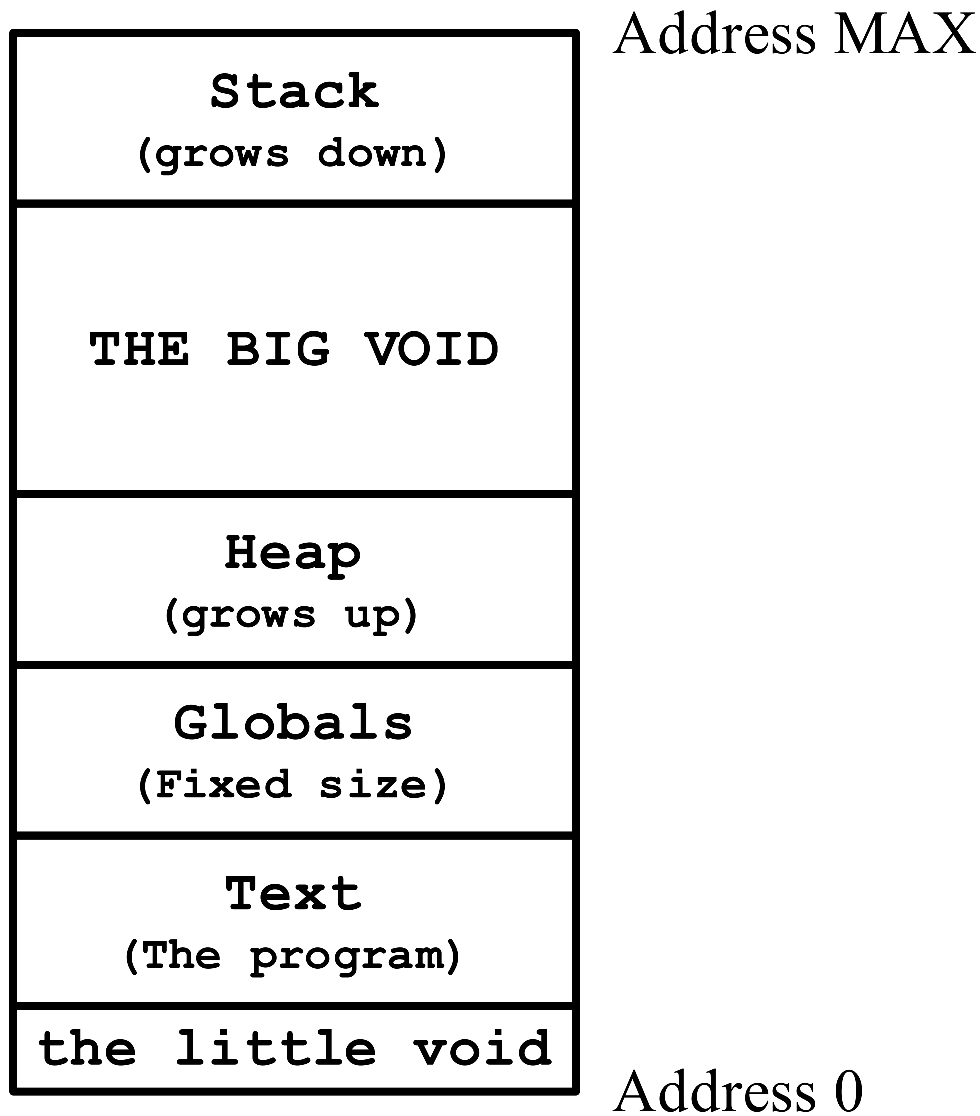
- When functions are called, stack frames are created on the *stack*, which grows downward.

- Since we don't know how big either of these will get, we keep a big hole in between the two called THE BIG VOID.



The little void

- Most systems also reserve the first few thousand addresses starting at zero for another void.

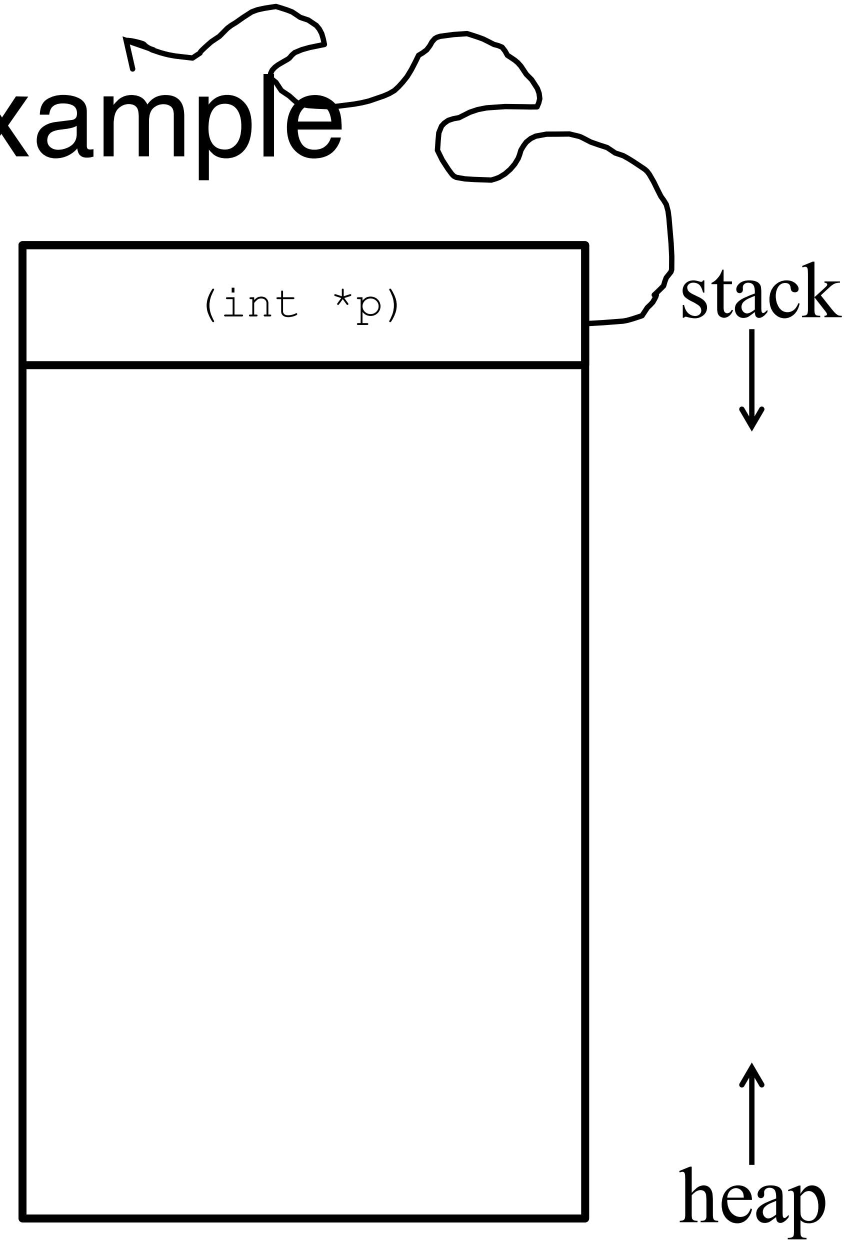


Global vs. local vs. dynamic

	Global	Local	Dynamic
Where in code?	Outside function	Inside function (block) or args	Anywhere you use `new`
When created	Beginning of program	Beginning of function (block)	You call `new`
When destroyed	End of program	End of function (block)	You call `delete`
Size	static	static	dynamic
Location	Globals	Stack	Heap

Stack and heap example

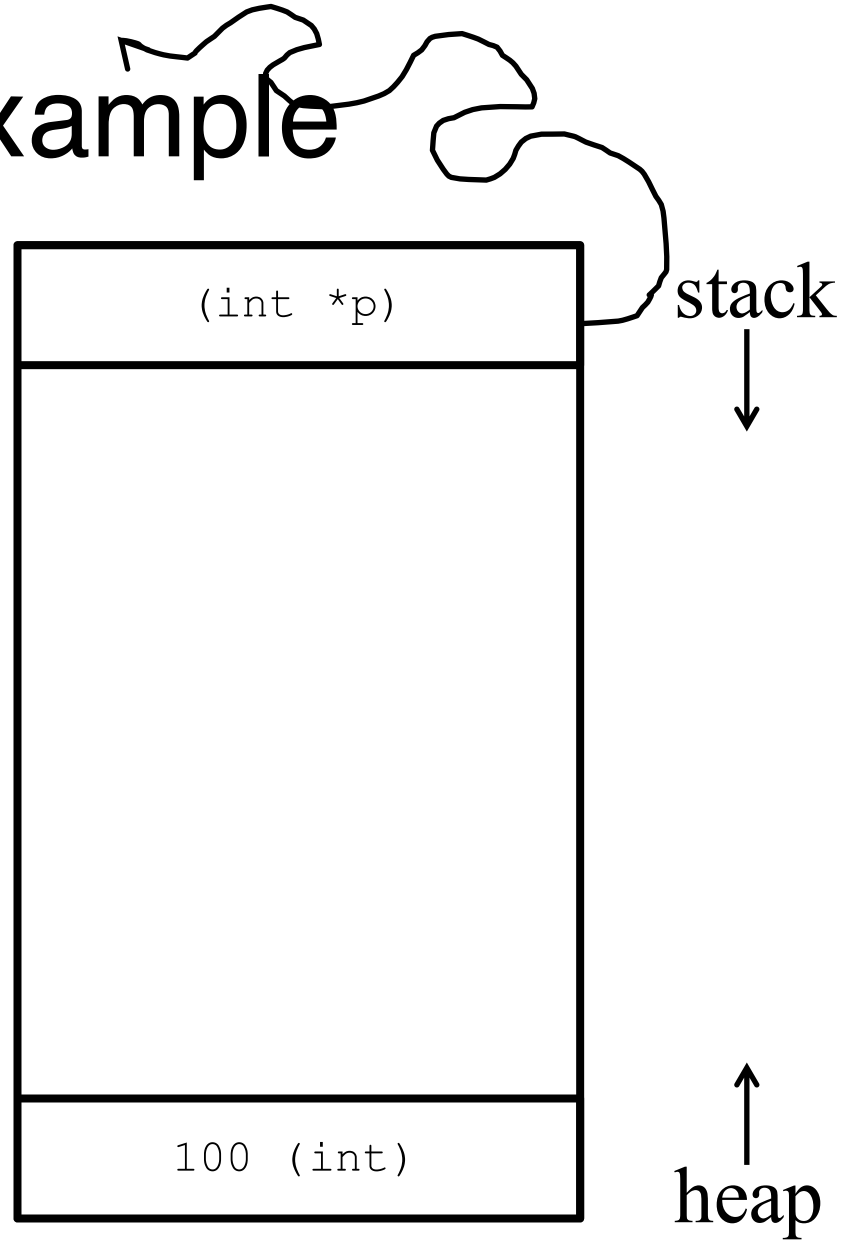
```
int main() {
  int *p;
}
```



- Local variable goes on the stack
- Undefined value
  - An address, since `p` is a pointer
- Could point to a random, part of memory

Stack and heap example

```
int main() {
  int *p;
  new int(100);
}
```

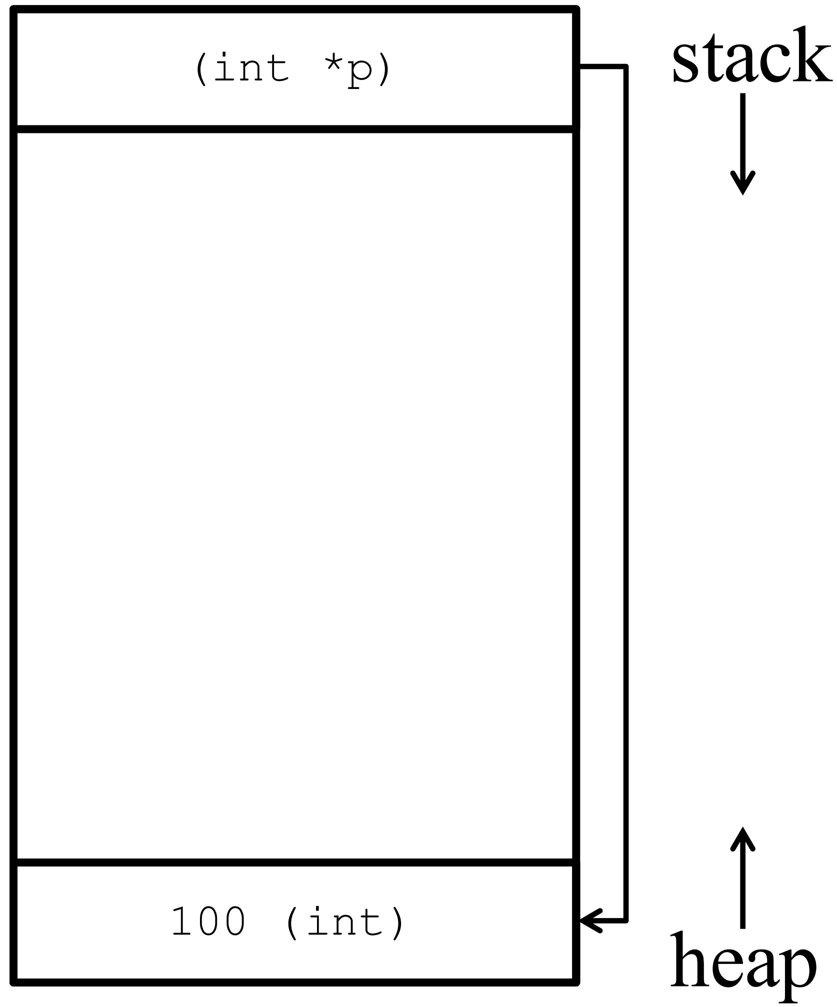


- Dynamic memory goes on the heap
- Leaks `sizeof(int)` bytes of memory
- No way to find `int(100)` and delete it!

Stack and heap example

```
int main() {
  int *p = new int(100);
}
```

- Good habit: always initialize variables

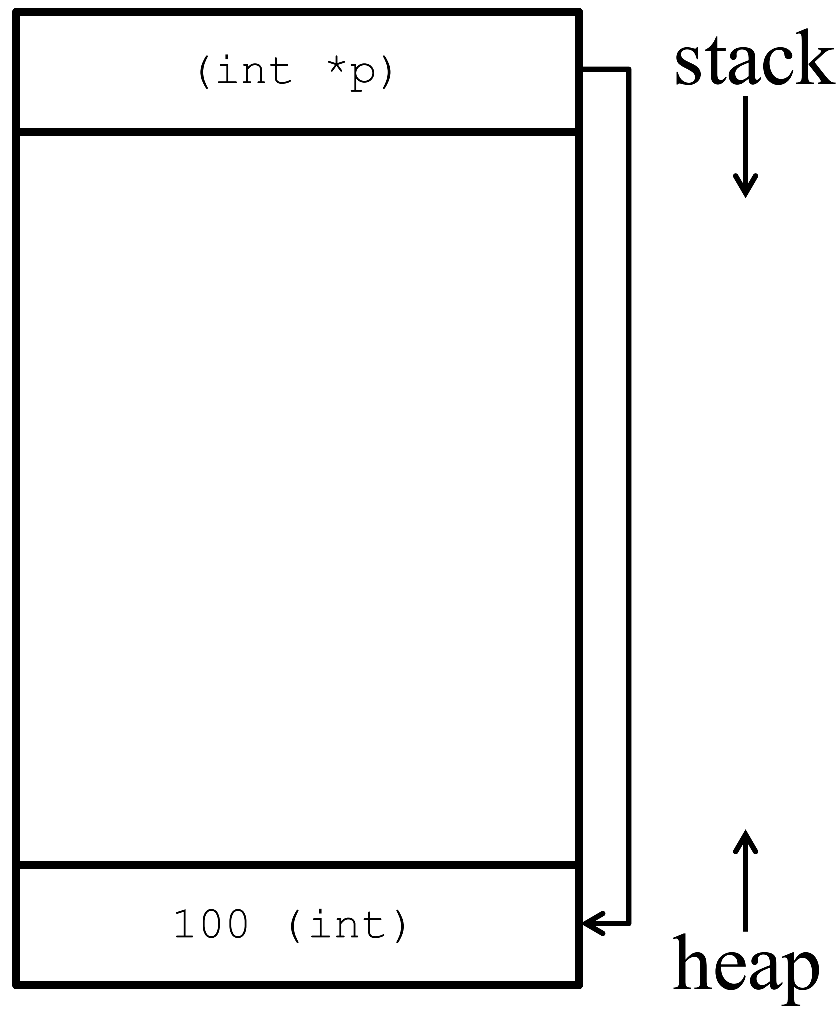


Stack and heap example

```
int main() {
  int *p = new int(100);
  cout << *p << endl;
}
```

```
100
```

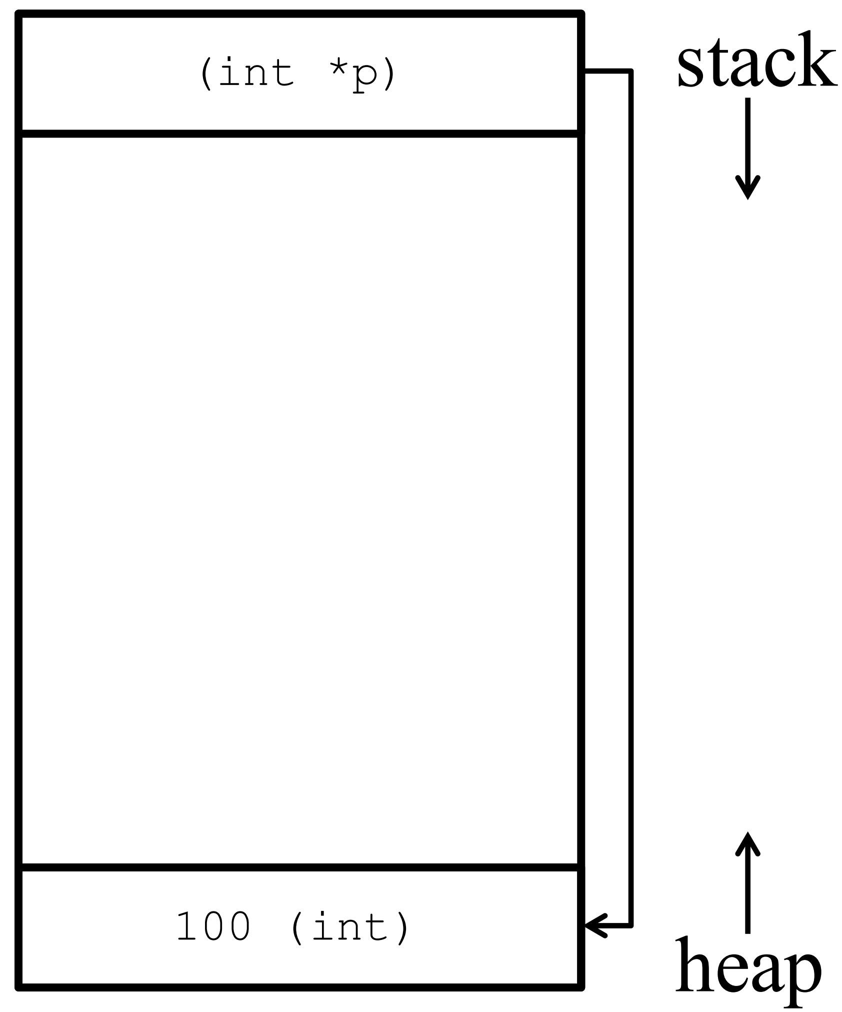
- Access dynamic variable using pointer



Stack and heap example

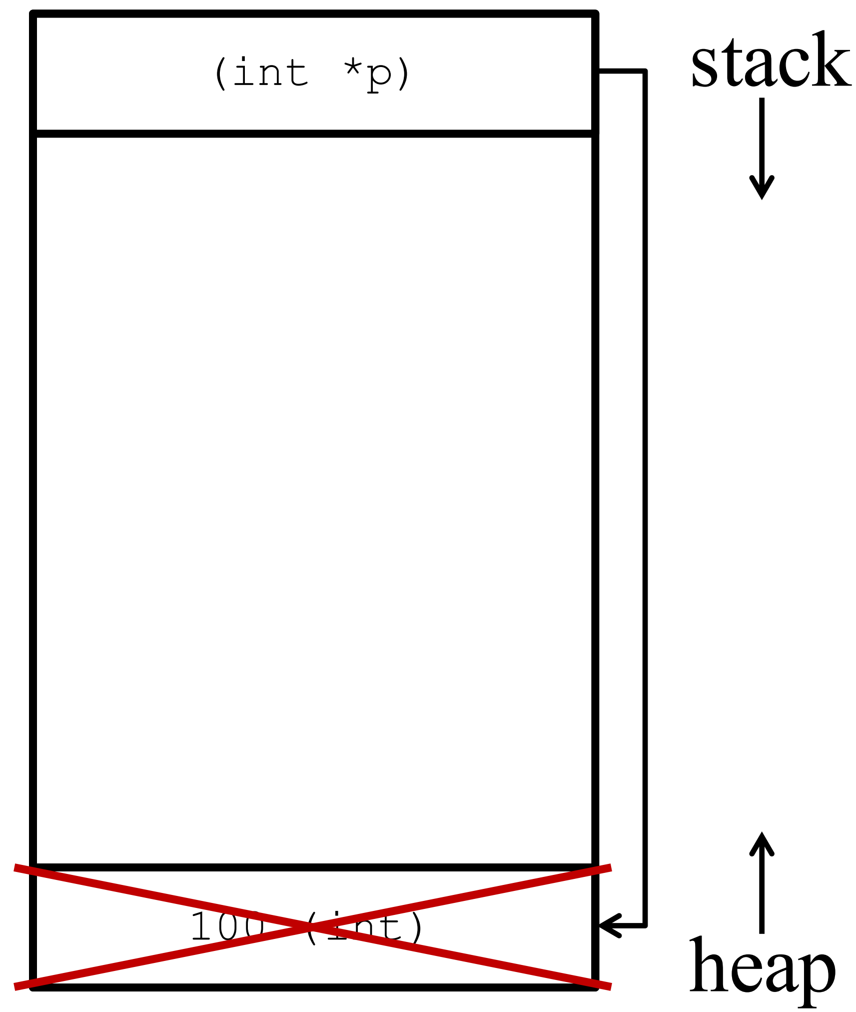
```
int main() {
  int *p = new int(100);
  cout << *p << endl;
}
```

- Problem: memory leak!



Stack and heap example

```
int main() {
  int *p = new int(100);
  delete p;
  cout << *p << endl;
}
```

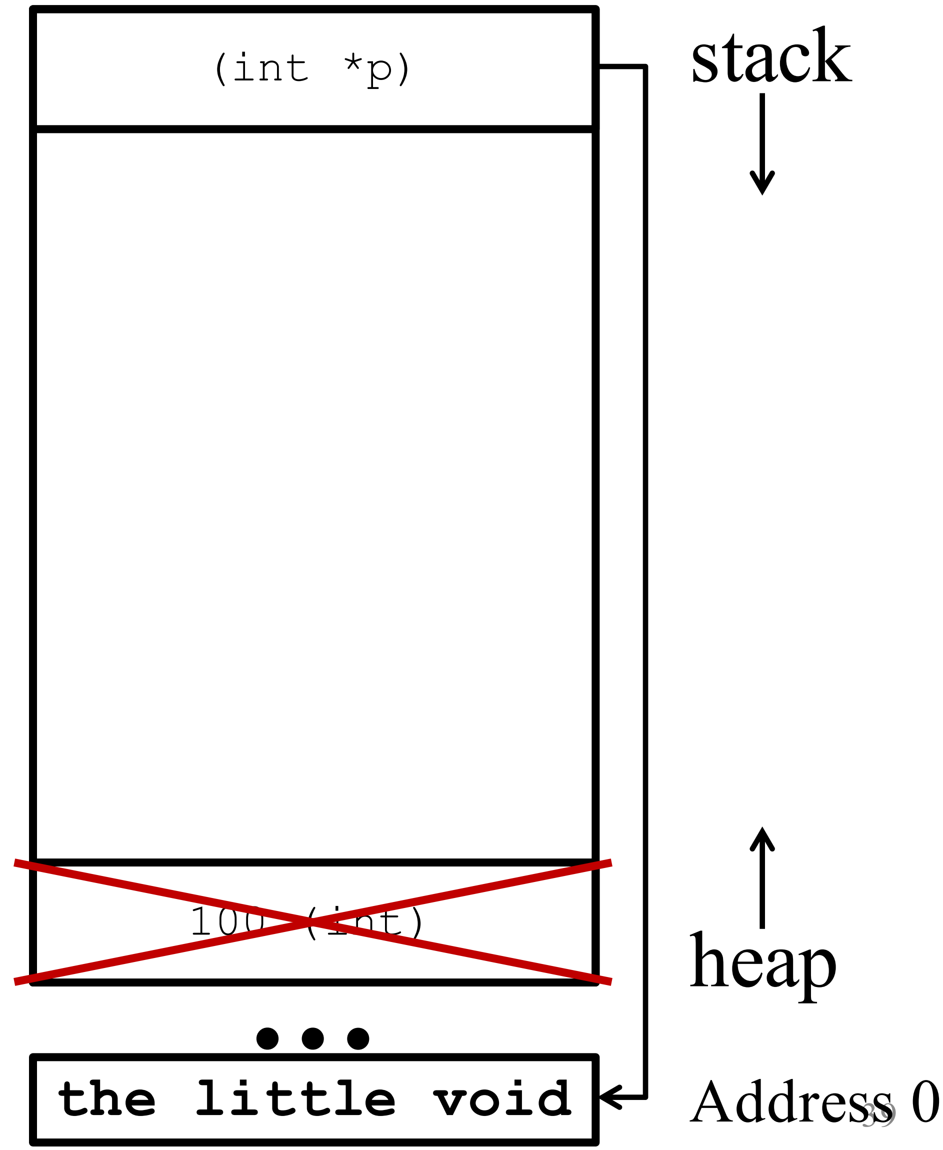


- Problem: memory leak!
- Fixed
- Problem: reading dynamic variable after `delete`

Stack and heap example

```
int main() {
  int *p = new int(100);
  delete p; p=0;
  cout << *p << endl;
}
```

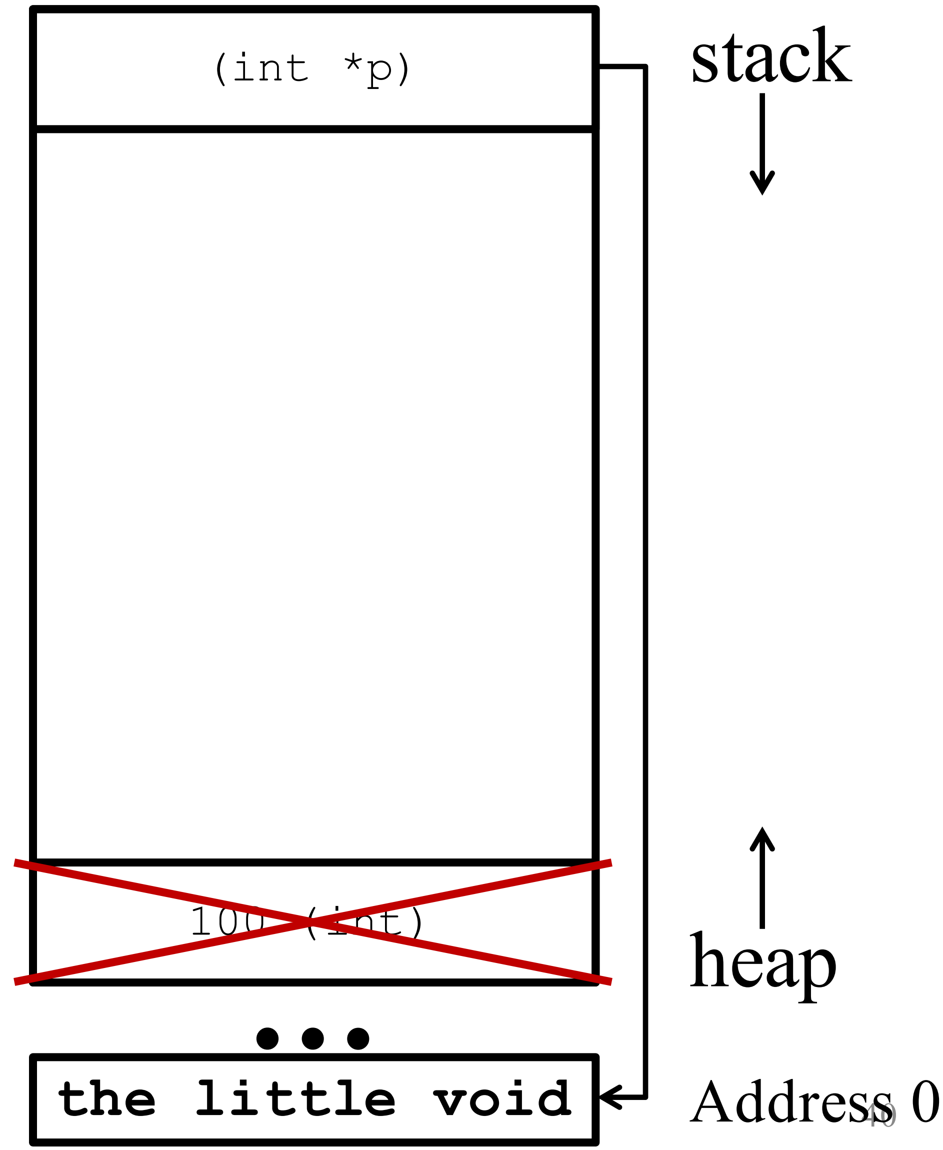
- Problem: reading dynamic variable after `delete`
- Fixed
- Problem: dereferencing a `0` (AKA `NULL`) pointer!



Stack and heap example

```
int main() {
  int *p = new int(100);
  delete p; p=0;
  assert(p);
  cout << *p << endl;
}
```

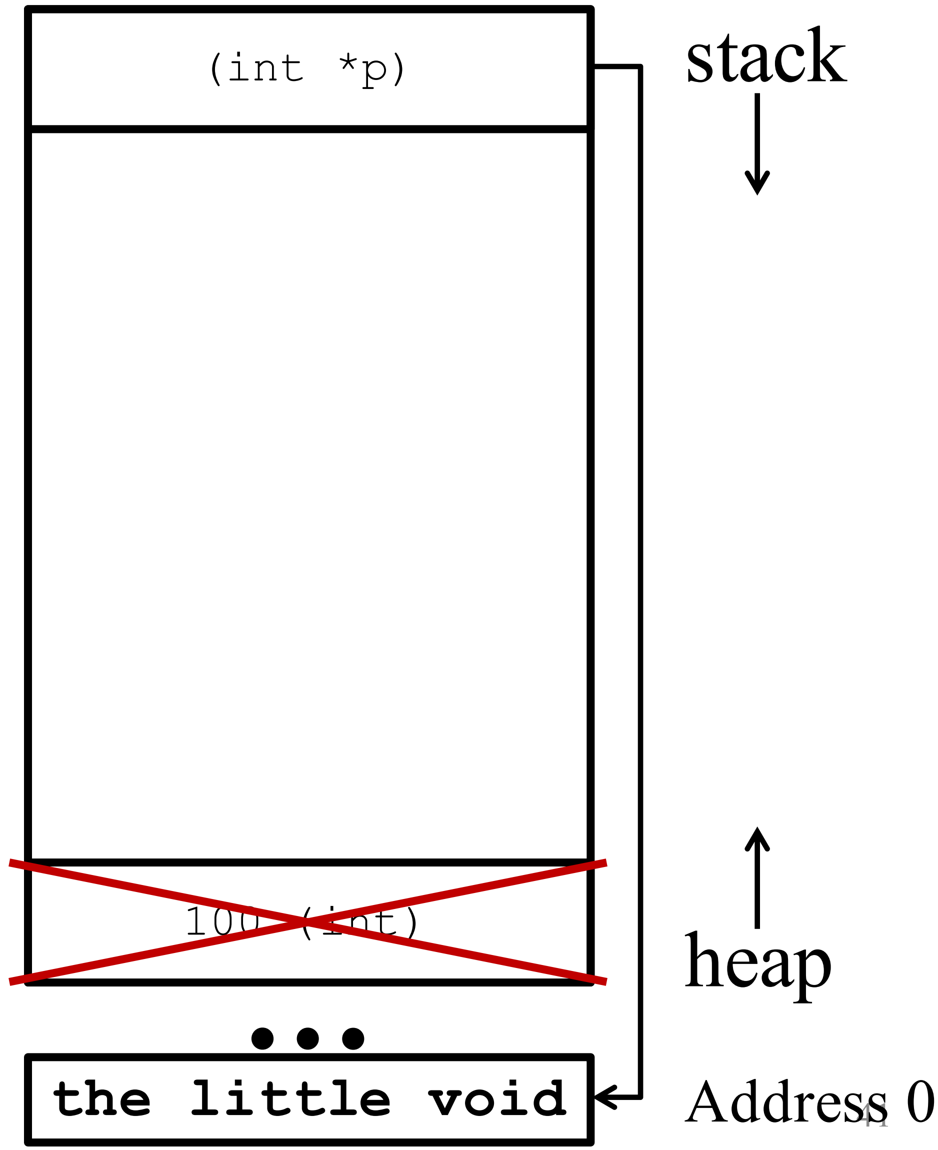
- Problem: dereferencing a `0` (AKA `NULL`) pointer!
- `assert()` can help identify problem before it causes a `SEGFAULT`



Stack and heap example

```
int main() {
  int *p = new int(100);
  assert(p);
  cout << *p << endl;
  delete p; p=0;
}
```

- Bugs fixed



NULL vs. 0

`int *ptr = NULL;`

- C style
- No library needed
- `int *ptr=0;`
  - Works, but `ptr=NULL` is preferred

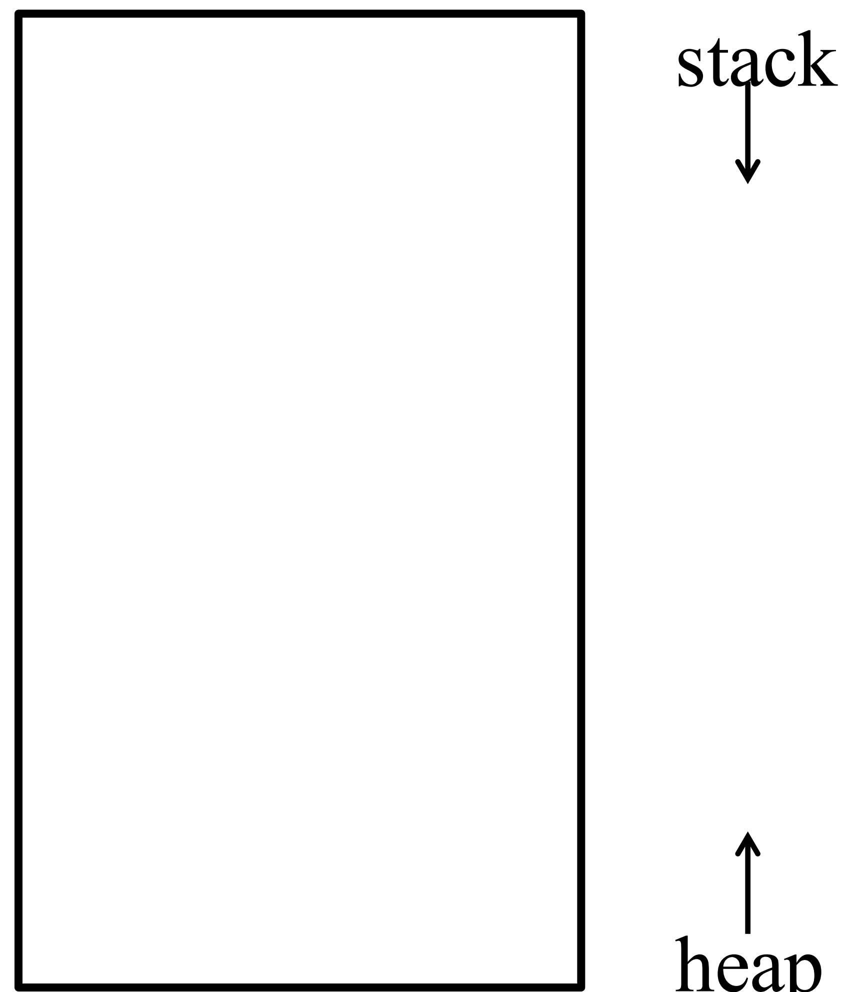
`int *ptr = 0;`

- C++ style
- `int *ptr=NULL;`
  - *error: 'NULL' was not declared in this scope*
- `#include <cstddef>`
  `int *ptr=NULL;`
  - Works, but `ptr=0` is preferred

Stack and heap exercise

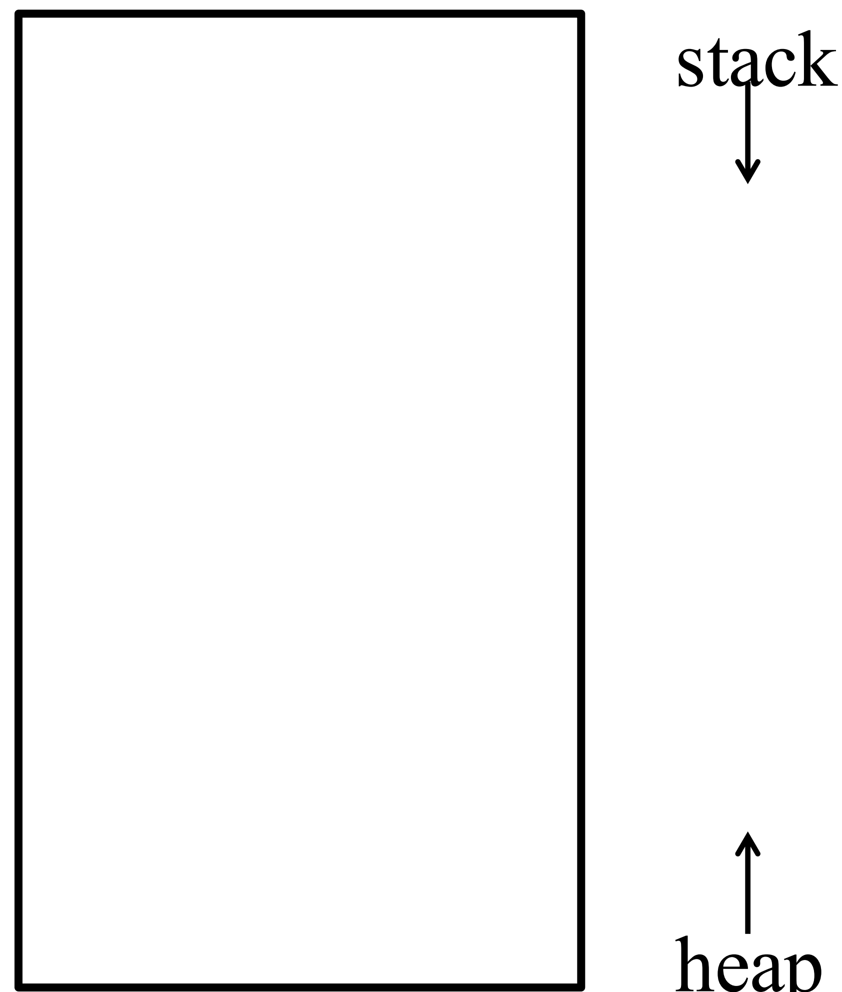
```
int i = 42;
int *p = &i;
delete p; p=0;
```

- Draw the stack and the heap
- What is wrong with this code?



Stack and heap exercise

```
int i=4;
int *p = new int(17);
i = *p;
delete p; p=0;
```

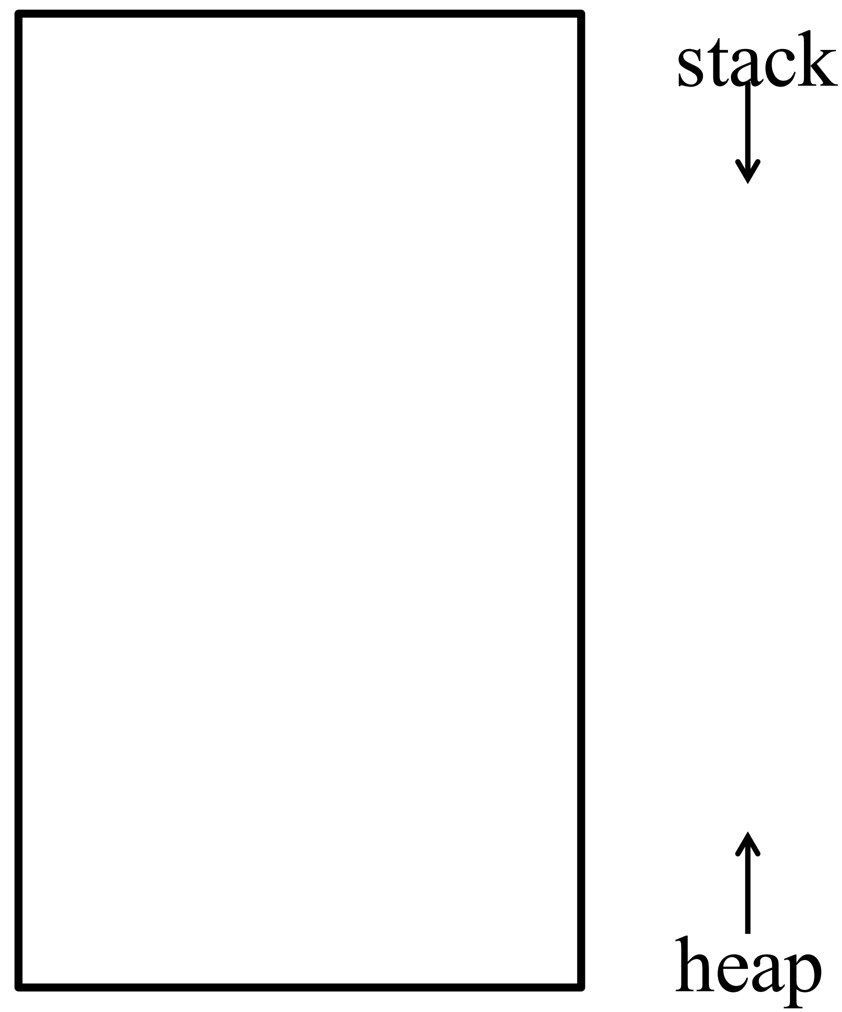


- Draw the stack and the heap
- How much memory is leaked?

Stack and heap exercise

```
int *p = new int(100);
int *q = p;
delete q; q=0;
cout << *p << endl;
```

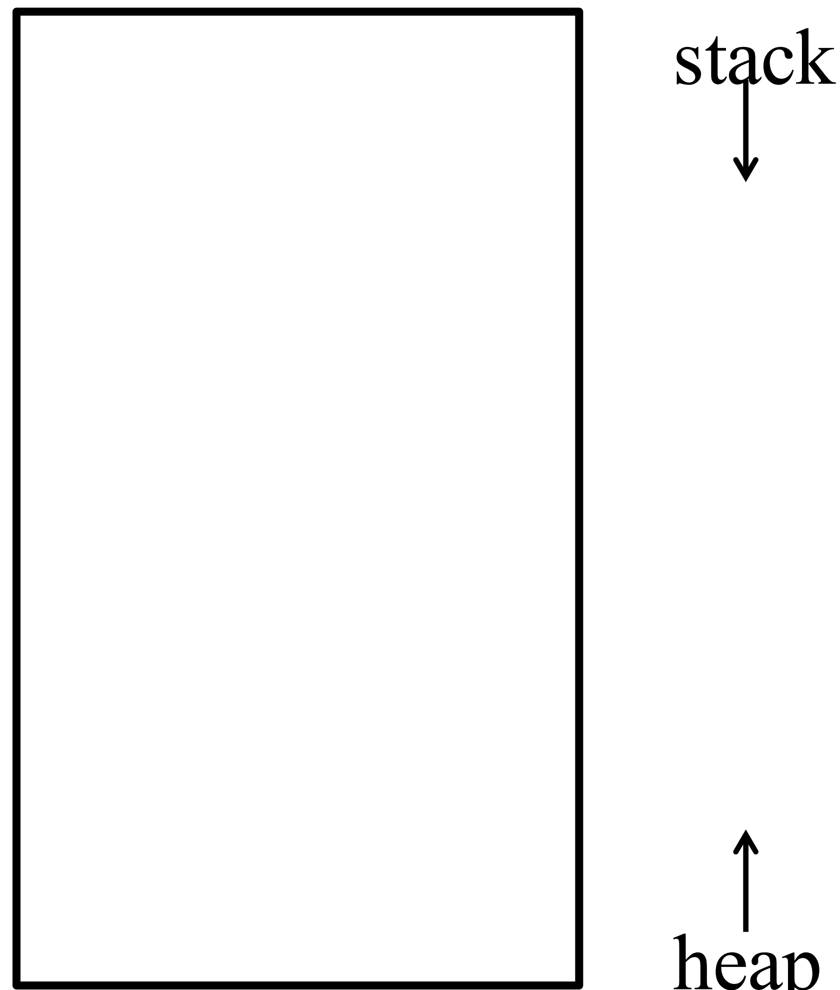
- Draw the stack and the heap
- What does this print?
- How much memory is leaked?



Stack and heap exercise

```
int *p = new int(100);
int *q = new int(42);
q=p;
delete q; q=0;
```

- Draw the stack and the heap
- How much memory is leaked?



Classes and dynamic memory

- When you create instances of classes, their constructors are called, just as if it were created the "normal" way.

```
IntSet *isp = new IntSet;
```

1. Allocate enough space on the heap to hold an `IntSet`.
   - An array of 100 integers (`elts`)
   - One extra integer (`elts_size`) to hold its size
2. Call the constructor `IntSet::IntSet()` on this new object

Classes and dynamic memory

- We can also destroy instances of ADTs that were created by new:

```
IntSet *isp = new IntSet;
delete isp; isp=0;
```

Exercise: allocating classes

```
IntSet *isp =
  new IntSet;
delete isp; isp=0;
```

- Draw the stack and heap
- Assume no virtual functions