Dynamic memory

Michael R. Nowak

michael@nowakphd.com

Anatomy of a program in memory

Code / Static Data	Where the code to be executed and other static data (think global variables, things explicitly tagged with the static keyword, etc.) are stored; lifetime of static data objects: throughout program execution
Heap / Free Store	The dynamic memory area, where dynamic objects created are stored; lifetime of heap objects: until explicitly deleted or when the program terminates
	In classical architectures, the stack and heap grow towards one another
Stack	Stores local variables, manages function calls; extensively involved in performing computations; lifetime of 'automatic' objects: persistent until the end of the block that declared them

*** Note: This is a simplified model*

Memory allocation

Stack

Allocation of memory to variables on the stack

- The size of variables must be known at compile-time
- A new block of memory called a stack frame (aka an activation record) is added to the stack to hold automatic variables each time you call a method

Heap / Free Store

Dynamic allocation of objects on the free store

- Size of objects may be unknown at compile-time
- Allocation performed at run-time
- Dynamically created objects are stored on the free store

Static

Allocation of memory for variables declared as static and global variables

- To be discussed at a later time

Allocation of memory to variables

Stack

- We can declare an `int` variable identified by `k` and initialized with 11 by writing the following code:

```
int k = 11;
```

- When the compiler observes this statement, it will
  - Determine an amount of memory to hold the value of an `int`
  - Will add the identifier `k` to a symbol table along with the relative memory address in which the object will become accessible during runtime
- As the thread of execution passes over this declaration, the value 11 will be placed in a memory location reserved for storage of `k`'s value

Allocation of memory to variables

Stack

- The size of objects stored on the stack must be known at compile-time, *why*?
- Recall that when a function (such as main) calls another function, an activation record for the called function is added to the top of the stack
  - When a function is called, the stack pointer is moved in one direction to allocate memory on the stack for the local variables associated with the called function
  - When the function finishes execution, the stack pointer is moved back in the other direction; memory is deallocated

Allocation of memory to variables

Stack | Activity

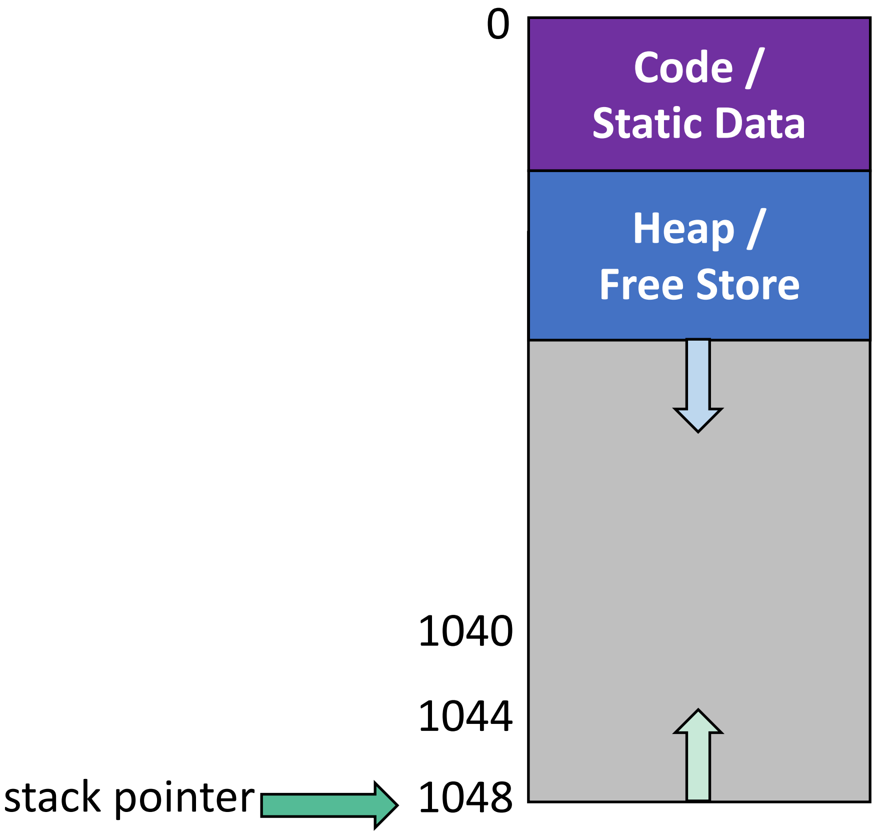
- What does the memory diagram look like for the following "application"?

```
int times2(int t1)
{
   int r1 = t1 * 2;
   return r1;
}

int main ()
{
   int i = 10;
   int z = times2(i);
}
```

Allocation of memory to variables

Stack



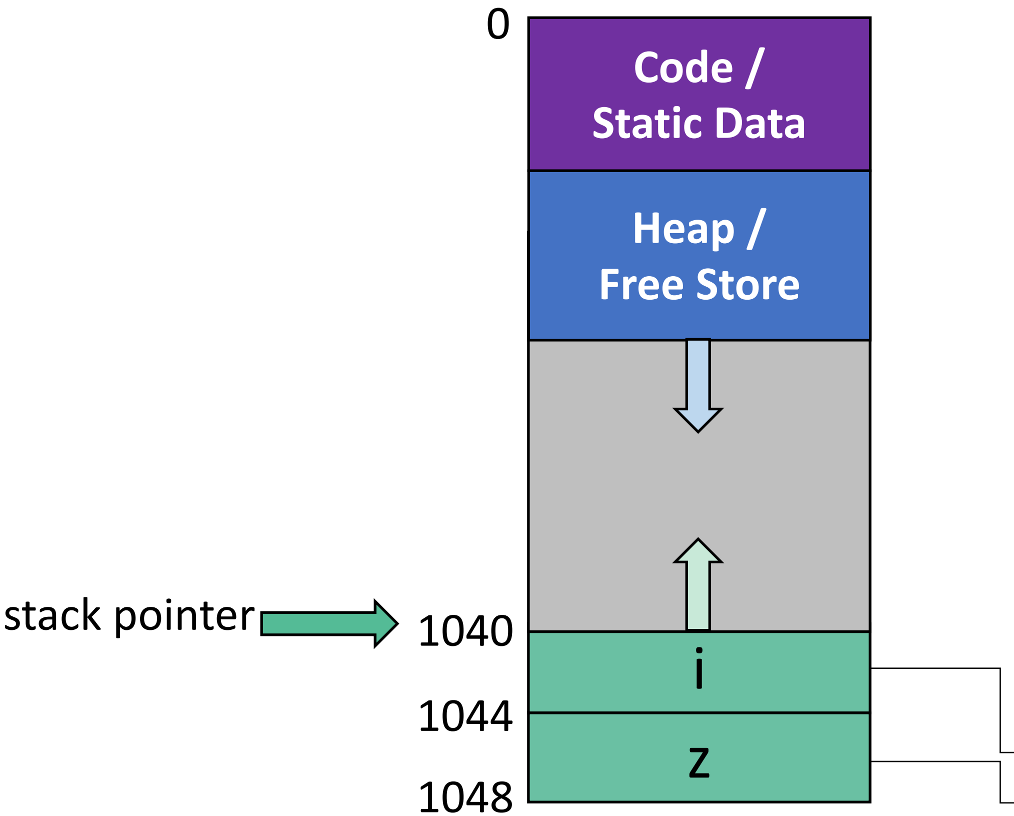

```
int times2(int t1)
{
   int r1 = t1 * 2;
   return r1;
}

int main ()
{
   int i = 10;
   int z = times2(i);
}
```

This model illustrates the classical architecture layout, where the stack and heap grow towards one another: as presented, the heap grows from lower address space to higher; meanwhile, our stack grows from higher addresses to lower addresses

Allocation of memory to variables

Stack



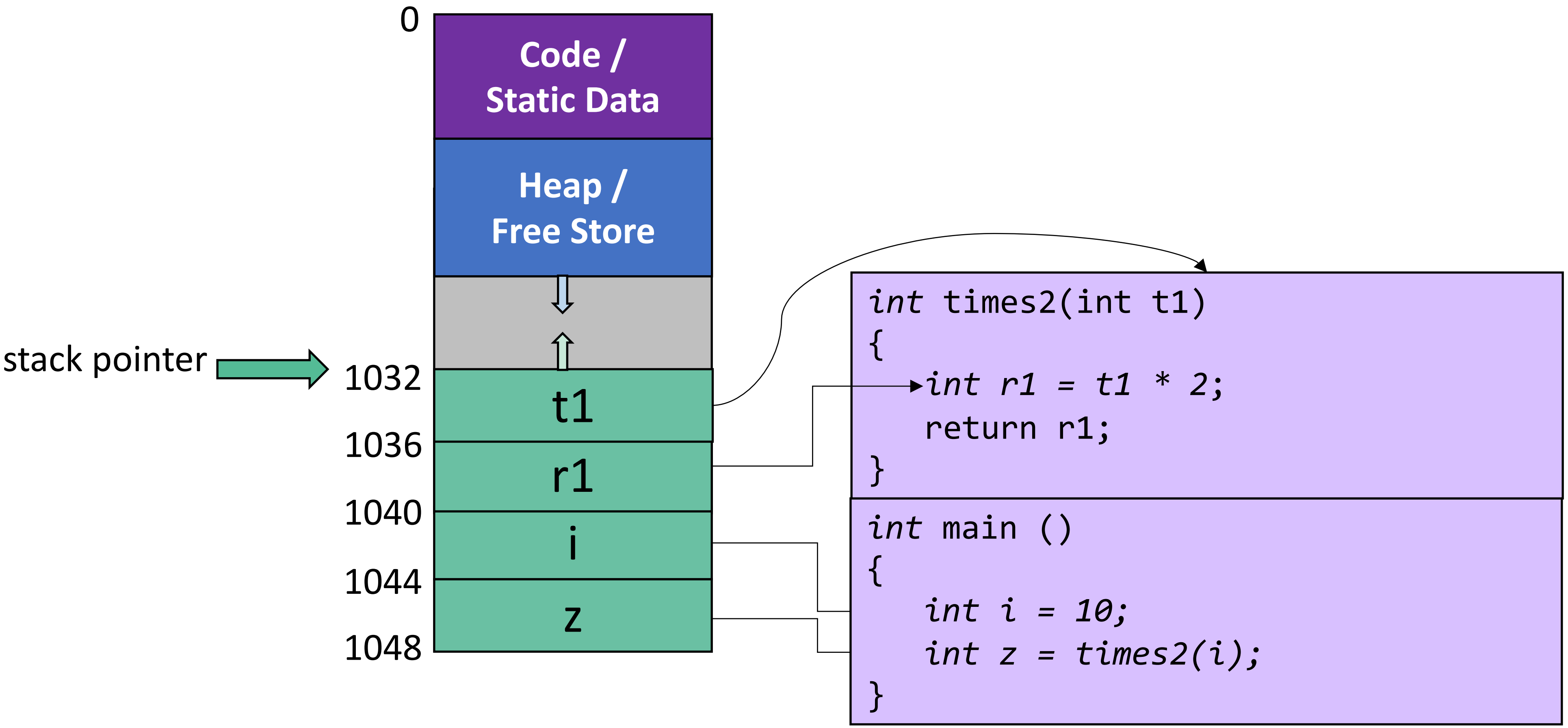

```
int times2(int t1)
{
   int r1 = t1 * 2;
   return r1;
}
```

```
int main ()
{
   int i = 10;
   int z = times2(i);
}
```

This model illustrates the classical architecture layout, where the stack and heap grow towards one another: as presented, the heap grows from lower address space to higher; meanwhile, our stack grows from higher addresses to lower addresses

Allocation of memory to variables

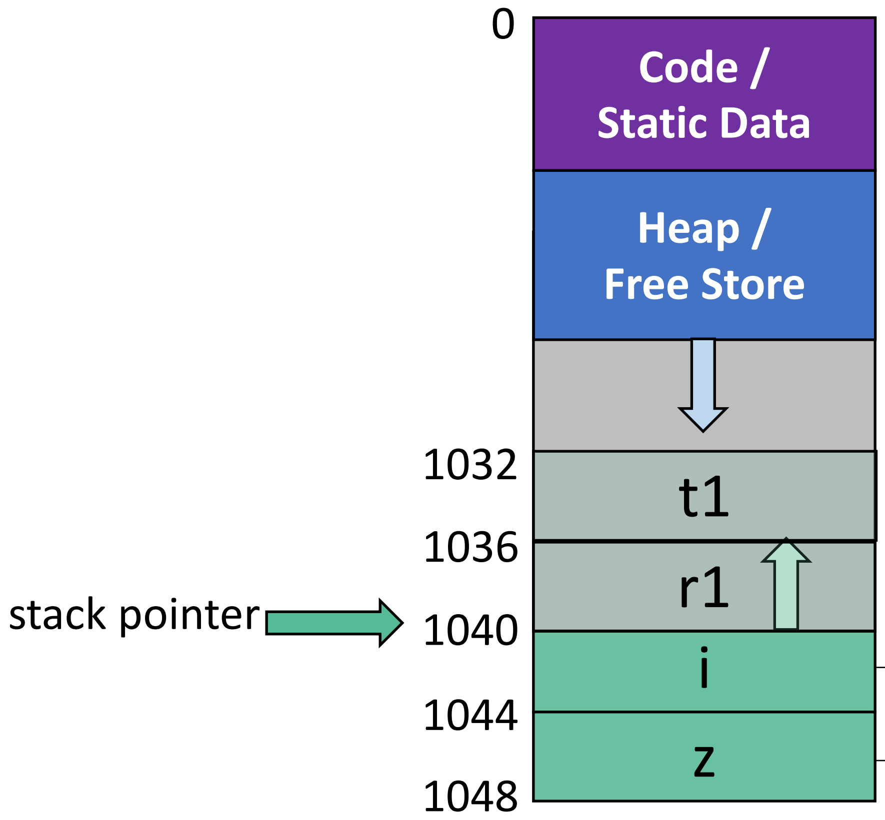
Stack



This model illustrates the classical architecture layout, where the stack and heap grow towards one another: as presented, the heap grows from lower address space to higher; meanwhile, our stack grows from higher addresses to lower addresses

Allocation of memory to variables

Stack



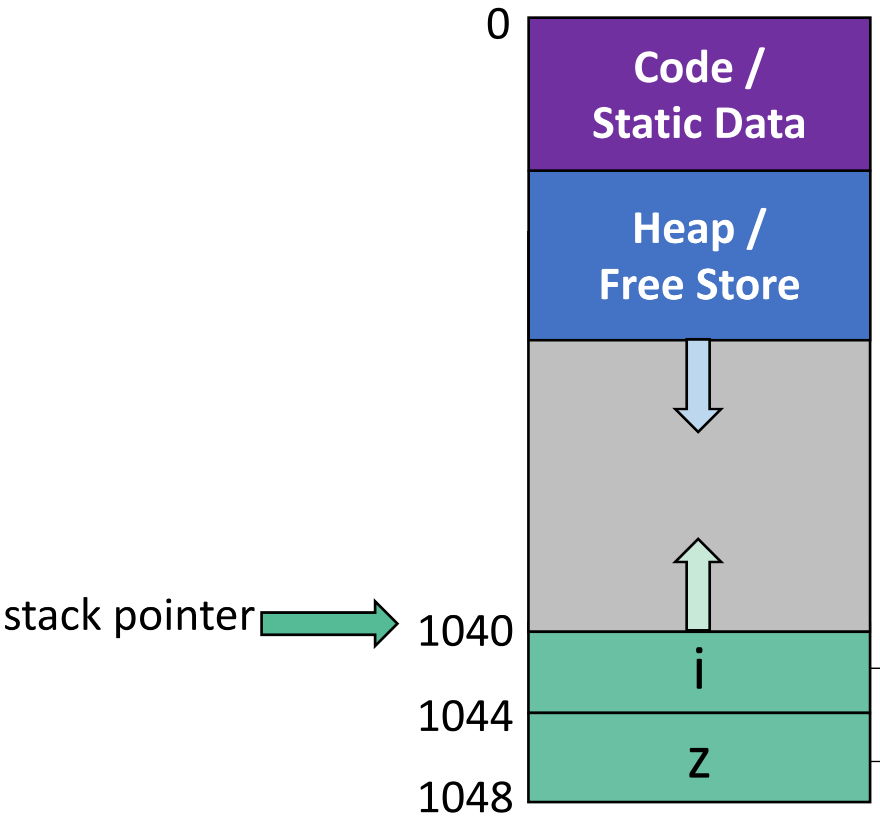

```
int times2(int t1)
{
   int r1 = t1 * 2;
   return r1;
}
```

```
int main ()
{
   int i = 10;
   int z = times2(i);
}
```

This model illustrates the classical architecture layout, where the stack and heap grow towards one another: as presented, the heap grows from lower address space to higher; meanwhile, our stack grows from higher addresses to lower addresses

Allocation of memory to variables

Stack




```
int times2(int t1)
{
   int r1 = t1 * 2;
   return r1;
}
```

```
int main ()
{
   int i = 10;
   int z = times2(i);
}
```

This model illustrates the classical architecture layout, where the stack and heap grow towards one another: as presented, the heap grows from lower address space to higher; meanwhile, our stack grows from higher addresses to lower addresses

Allocation of memory to variables

Stack

- If the sizes of the local variables, etc. are fixed at compile-time
- And if these local variables are stored in the same order in the activation record each time that the function is invoked
- Then the location of each local variable will always be at a fixed offset from the stack pointer
- For instance, when an invocation of `times2(int)` is being executed

```
int times2(int t1)
{
    int r1 = t1 * 2;
    return r1;
}
```

`t1` may then *always* be accessed through `StackPtr + 0` and `r1` through `StackPtr + sizeof(int)`

Allocation of memory to variables

Stack

- So why must the size of variables stored on the stack be known at compile-time?
  - If we were to introduce a variable length array into this mix
  - Then those offsets into the local variables will no longer be fixed at compile time
  - Instead, they (i.e., the offsets) will become dependent on the size of the array used in the particular invocation of the function
  - And well, this complicate things for the compiler designer

Allocation of memory to variables

Stack

Now consider the following definitions of func2a() and func2b():

```
void func2a(int size)
{
    int a[3];
    double c;
}
```

size may *always* be accessed through StackPtr + 0 , a may *always* be accessed through StackPtr + sizeof(int) and c through StackPtr + sizeof(int) + sizeof(int) * 3

```
void func2b(int size)
{ // Note: variable length arrays are not apart of the C++14 standard
    int a[size];
    double c;
}
```

size may *always* be accessed through StackPtr + 0 , a may *always* be accessed through StackPtr + sizeof(int) and c through StackPtr + sizeof(int) + sizeof(int) * size

Allocation of memory to variables

Stack

Now consider the following definitions of func2a() and func2b():

```
void func2a(int size)
{
    int a[3];
    double c;
}
```

```
void func2b(int size)
{ // Note: variable length arrays are not apart of the C++14 standard
    int a[size];
    double c;
}
```

`size` may *always* be accessed through StackPtr + 0 , `a` may *always* be accessed through StackPtr + sizeof(int) and `c` through `StackPtr + sizeof(int) + sizeof(int) * 3`

`c` is **always** at the same offset from the `StackPtr`!

`size` may *always* be accessed through StackPtr + 0 , `a` may *always* be accessed through StackPtr + sizeof(int) and `c` through `StackPtr + sizeof(int) + sizeof(int) * size`

`c` is **not always** at the same offset from the `StackPtr`!

The C++ standard states that an array's size must be a constant expression (8.3.4.1)

- Given that variable length arrays (VLAs) on the stack are not in the standard (as of C++14), you might question why this compiles for you (and not for the person sitting next to you)

```
#include <iostream>
using namespace std;

int main(int argc, char * argv[])
{
    int size;
    cout << "Enter a size for the array : ";
    cin >> size;
    int array[size];
    return 0;
}
```

The C++ standard states that an array's size must be a constant expression (8.3.4.1)

- VLAs on the stack are valid in the C99 standard (we're talking *C* here, *not C++*)
- The reason why VLAs on the stack may work for you on one system and not another, is that some C++ compilers have chosen to support VLAs on the stack
    - Given that VLAs are not included in the C++ standard, I wouldn't recommend using them
    - In fact, I would recommend using a `vector` over an array whenever practical

Allocation of memory to variables

Stack | Activity

- What's "problematic" about `problematic()`? Draw a memory diagram to illustrate.

```
int* problematic()
{
    int data[4] {10, 20, 30, 40};
    return data;
}

int main()
{
    int *my_data = problematic();

    return 0;
}
```

Memory allocation

Stack

Allocation of memory to variables on the stack

- The size of variables must be known at compile-time
- A new block of memory called a stack frame (aka an activation record) is added to the stack to hold local variables each time you call a method

Heap / Free Store

Dynamic allocation of objects on the free store

- Size of objects may be unknown at compile-time
- Allocation performed at run-time
- Dynamically created objects are stored on the free store

Static

Allocation of memory for variables declared as static and global variables

- To be discussed at a later time

Dynamically allocated objects

Heap / Free Store

- We cannot create new variables (named objects) at run-time, but we can create dynamically allocated objects while the program is running
- The C++ runtime maintains a pool of memory called the free store
- We can dynamically allocated memory on the free store in C++ with the **`new`** operator:
  - The new operator allocates memory for an object (or array of objects) of a specified type on the free store
  - The new operator returns the address of the region of memory allocated for that object
- It is *our responsibility* to explicitly deallocate that memory when we are finished using it; we do this by using the address as an operand to the **`delete`** operator

Dynamically allocated objects

Heap / Free Store

Allocation of dynamic memory

- To dynamically allocate an object, we use the new operator, followed by the type of object being dynamically allocated and store the address returned by the new operator in a pointer

```
int *p_int = new int;
```

```
        p_int                      *p_int
 .---------------.          .---------------.
 | 0x7f95a1c04d50|------>|        0       |
 '---------------'          '---------------'
 ( 0x7fff5dcfa2e8)          ( 0x7f95a1c04d50)
```

```
double *p_double = new double;
```

```
       p_double                   *p_double
 .---------------.          .---------------.
 | 0x7fa6e2500010|------>|    0.000000   |
 '---------------'          '---------------'
 ( 0x7fff5b9702e0)          ( 0x7fa6e2500010)
```

Dynamically allocated objects

Heap / Free Store

Accessing a dynamically created object

- To access an object that we've dynamically allocate space for, we we apply the dereference operator `*` to the pointer pointing to it

```
int *p_int = new int;
```

```
       p_int                     *p_int
 .---------------.          .---------------.
 | 0x7fa03a404d50|------->|       0       |
 '---------------'          '---------------'
 ( 0x7fff5a2682e8)          ( 0x7fa03a404d50)
```

```
*p_int = 5;
```

```
       p_int                     *p_int
 .---------------.          .---------------.
 | 0x7fa03a404d50|------->|       5       |
 '---------------'          '---------------'
 ( 0x7fff5a2682e8)          ( 0x7fa03a404d50)
```

Dynamically allocated objects

Heap / Free Store | Question

Deallocation of dynamic memory

- To deallocated memory that was allocated with new, we apply the delete operator to the pointer pointing to it

```
/* do something before */
int *p_int = new int;
*p_int = 5;

delete p_int;
p_int = nullptr;
/* do something else after */
```

- After you've deallocated memory, you should assign `nullptr` to the pointer. **Why bother?**

Dynamically allocated objects

Heap / Free Store | Question

In the following code, what's on the stack? What's on the free store?

```
int *p_array_int = new int[4];
// ... do something exciting
delete[] p_array_int;
p_array_int = nullptr;
```

Dynamically allocated objects

Heap / Free Store | Question

Do you notice anything different about the delete operator when applied to arrays compared to when we use it to deallocate objects of the primitive types?

```
int *p_array_int = new int[4];
// ... do something exciting
delete[] p_array_int;
p_array_int = nullptr;
```

```
int *p_int = new int;
// ... do something exciting
delete p_int;
p_int = nullptr;
```

Revisiting problematic()

Stack | Activity

- How do we "resolve" `problematic()`?

```
int* problematic()
{
    int data[4] {10, 20, 30, 40};
    return data;
}

int main()
{
    int *my_data = problematic();

    return 0;
}
```

Revisiting problematic()

Heap / Free Store | Question

- "Resolving" `problematic()`? **Use the free store!!**

```
int* problematic()
{
    int* data = new int[4] {10, 20, 30, 40};
    return data;
}

int main()
{
    int *my_data = problematic();

    return 0;
}
```

- Who's now responsible for deallocating this memory? The caller or callee?

Dynamically allocated objects

Heap / Free Store

An implicit contract

Clause	Result of violation
You will eventually return the memory that you borrow	Memory leak
You will immediately stop using the memory that you've returned	Dangling pointer
You will not return memory that you did not borrow (and you will not twice return memory that you've borrowed once)	Corrupted heap

Dynamically allocated objects

Heap / Free Store

Pitfalls of dynamic memory usage

- **Memory leak**
  - A memory leak occurs when we dynamically allocate memory for an object, but fail to ever deallocate that space when we're done using it
- **Dangling pointer**
  - A pointer to a piece of memory that has been deallocated is used
  - Until that memory is actually allocated again, you may continue to get the value that was stored in the object that use to reside there
- **Corrupted heap**
  - This can happen when something is deallocated that is not allocated

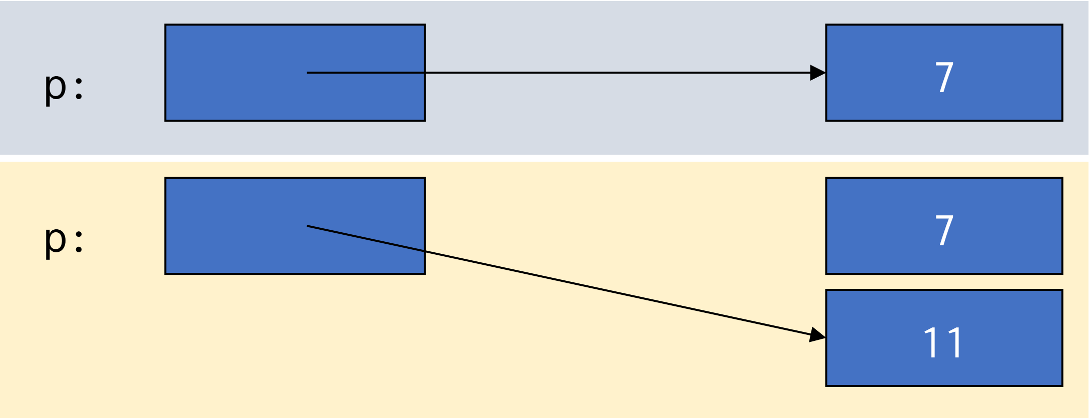
Dynamically allocated objects

Heap / Free Store

Pitfalls of dynamic memory usage : memory leaks

- **Memory leak**
  - A memory leak occurs when we dynamically allocate memory for an object, but fail to ever deallocate that space when we're done using it

```
int *p = new int(7);
/* do something with free store
object pointed to by p)*/
p = new int(11);
/* do something else */
delete p;
```




```
==48490== LEAK SUMMARY:
==48490==    definitely lost: 4 bytes in 1 blocks
==48490==    indirectly lost: 0 bytes in 0 blocks
==48490==      possibly lost: 0 bytes in 0 blocks
```

Dynamically allocated objects

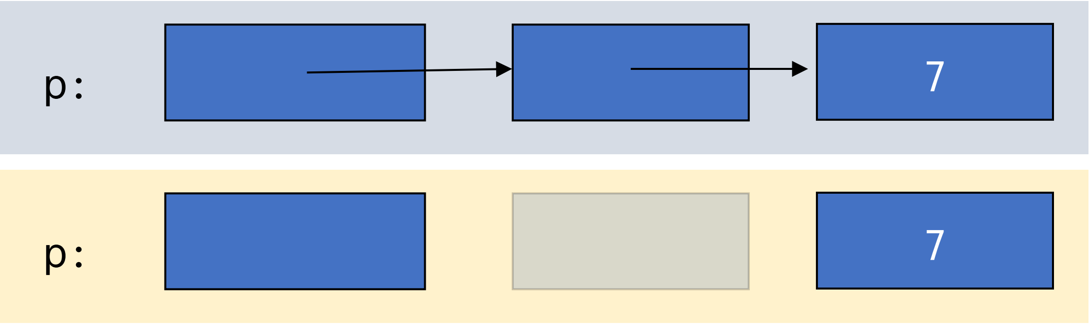
Heap / Free Store

Pitfalls of dynamic memory usage : memory leaks

- **Memory leak**
  - A memory leak occurs when we dynamically allocate memory for an object, but fail to ever deallocate that space when we're done using it

```
         p                        *p                      **p
.---------------.        .---------------.        .---------------.
| 0x7fe862500010|------->| 0x7fe862500000|------->|       7       |
'---------------'        '---------------'        '---------------'
( 0x7fff5423d318)        ( 0x7fe862500010)        ( 0x7fe862500000)
```

```
/* do something */
int **p = new int*(new int(7));
/* do something */
delete p; p = nullptr;
```




```
==48490== LEAK SUMMARY:
==48490==    definitely lost: 4 bytes in 1 blocks
==48490==    indirectly lost: 0 bytes in 0 blocks
==48490==      possibly lost: 0 bytes in 0 blocks
```

The **correct way** to deallocate memory in such instances as this is to: (1) delete the object to which p points; (2) delete p

```
delete *p; *p = nullptr;
delete p; p = nullptr;
```

```
==48458== LEAK SUMMARY:
==48458==    definitely lost: 0 bytes in 0 blocks
==48458==    indirectly lost: 0 bytes in 0 blocks
==48458==      possibly lost: 0 bytes in 0 blocks
```

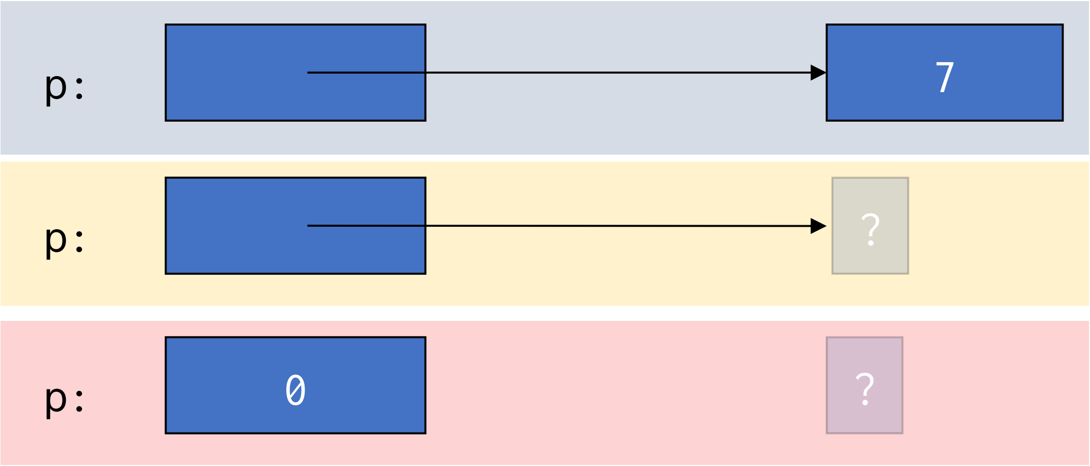
Dynamically allocated objects

Heap / Free Store

Pitfalls of dynamic memory usage : dangling pointer

- **Dangling pointer**
  - A pointer to a piece of memory that has been deallocated is used
  - Until that memory is actually allocated again, you may continue to get the value that was stored in the object that use to reside there

```
int *p = new int(7);
/* do something with free store
object pointed to by p */
/* somewhere else, create pointer
to that object */
delete p;
p = nullptr;
```



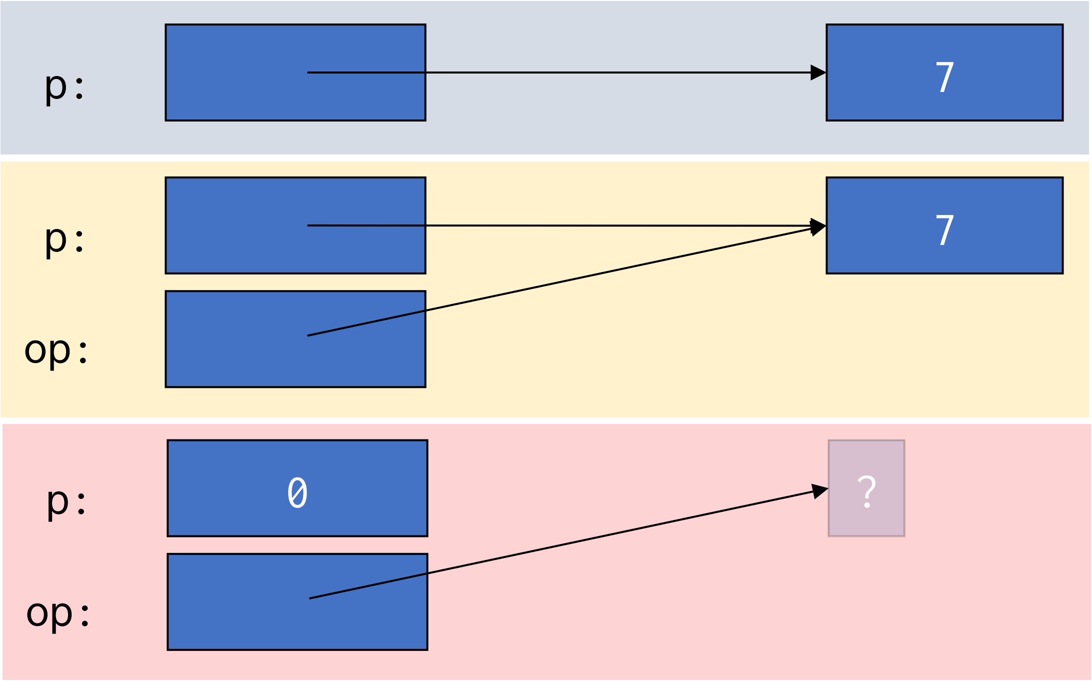
Dynamically allocated objects

Heap / Free Store

Pitfalls of dynamic memory usage : dangling pointer

- **Dangling pointer**
  - A pointer to a piece of memory that has been deallocated is used
  - Until that memory is actually allocated again, you may continue to get the value that was stored in the object that use to reside there

```
int *p = new int(7);
/* do something with free store
object pointed to by p */
/* somewhere else, create pointer
to that object */
int *op = p;
/* do something else */
delete p;
p = nullptr;
```



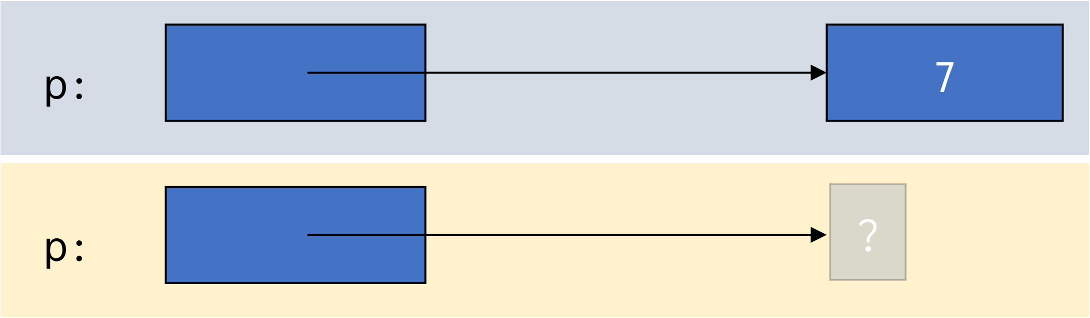
Dynamically allocated objects

Heap / Free Store

Pitfalls of dynamic memory usage : corrupted heap

- Corrupted heap
  - This can happen when something is deallocated that is not allocated

```
int *p = new int(7);
/* do something with free store
object pointed to by p */
delete p;
/* do something else */
delete p;
```




```
a.out(46378,0x7fff75151300) malloc: *** error for object
0x7fca28404d50: pointer being freed was not allocated
```

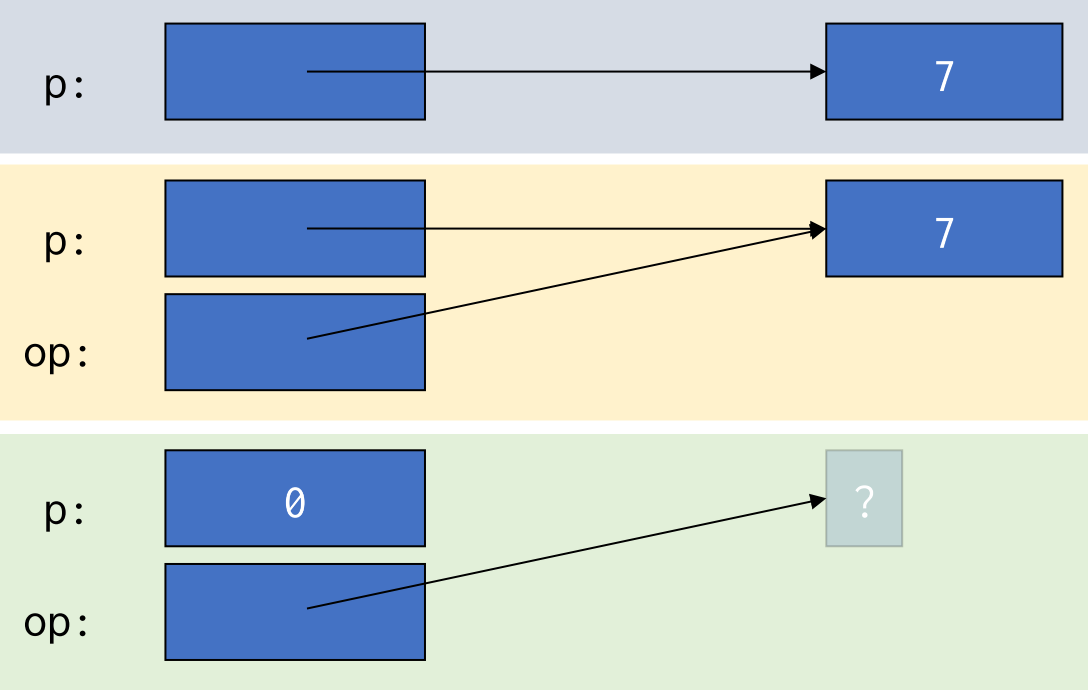
Dynamically allocated objects

Heap / Free Store

Pitfalls of dynamic memory usage : corrupted heap

- Corrupted heap
  - This can happen when something is deallocated that is not allocated

```
int *p= new int(7);
/* do something */
int *op= p;
/* do something */
delete p; p= nullptr;
/* do something */
delete op;
```




```
a.out(46378,0x7fff75151300) malloc: *** error for object
0x7fca28404d50: pointer being freed was not allocated
```

Wrap-up