Memory Management and Object Layout

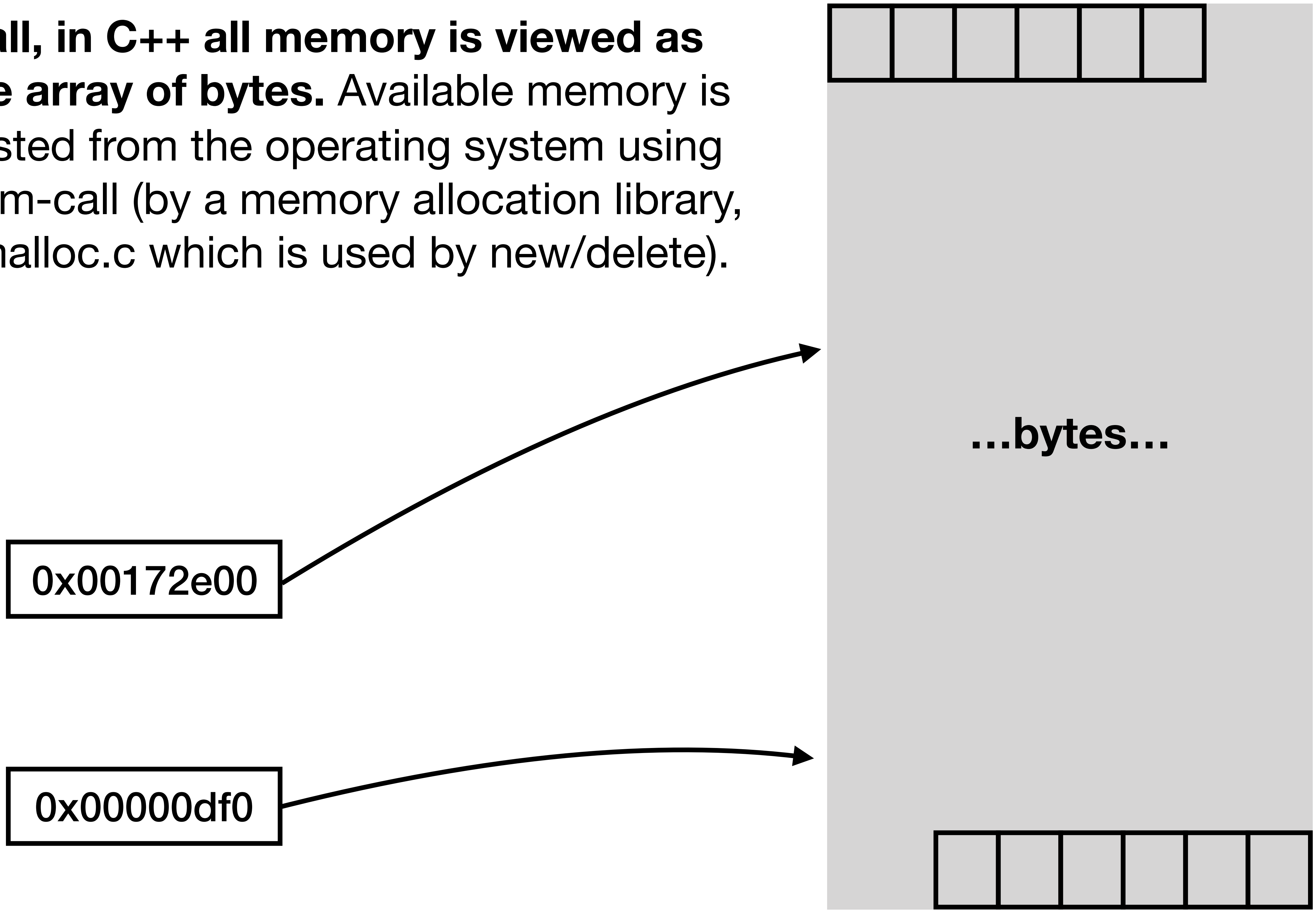
Logistics

- Lots of good questions on Slack so far
- Gone next Tuesday/Wednesday (will make video lecture)
- **Project 1 now up — Due next Tuesday**
- **Lab tomorrow**
- Project 2 up next Wednesday
  - Assembly language

Memory Management

C++ semantics: memory model

Recall, in C++ all memory is viewed as a huge array of bytes. Available memory is requested from the operating system using a system-call (by a memory allocation library, e.g., malloc.c which is used by new/delete).



C++ semantics: memory model

Recall, in C++ all memory is viewed a a huge array of bytes. Available memory is requested from the operating system using a system-call (by a memory allocation library, e.g., malloc.c which is used by new/delete).

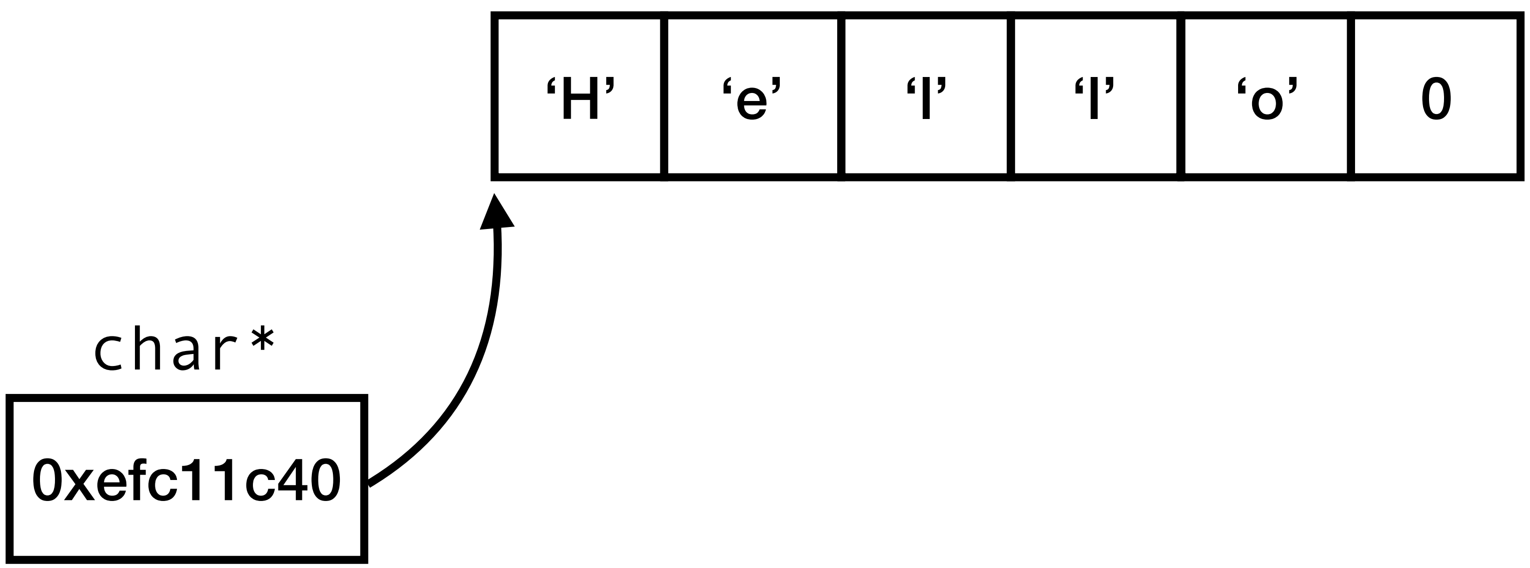
The stack starts growing down! The C++ runtime reserves a portion of memory (that is extended dynamically upon a page fault).

0x00172e00

The heap starts below the stack in memory and grows up, page by page.

0x00000df0

C++ semantics: memory model



Recall, in C++ pointers are (virtual) memory addresses and refer to the <u>start</u> of a buffer. Exactly how many bytes are being used by this pointer, after that location, is determined by how the C++ program uses that pointer! **(E.g., C-strings are null-value terminated.)** This is not statically checked, leading to buffer overflow.

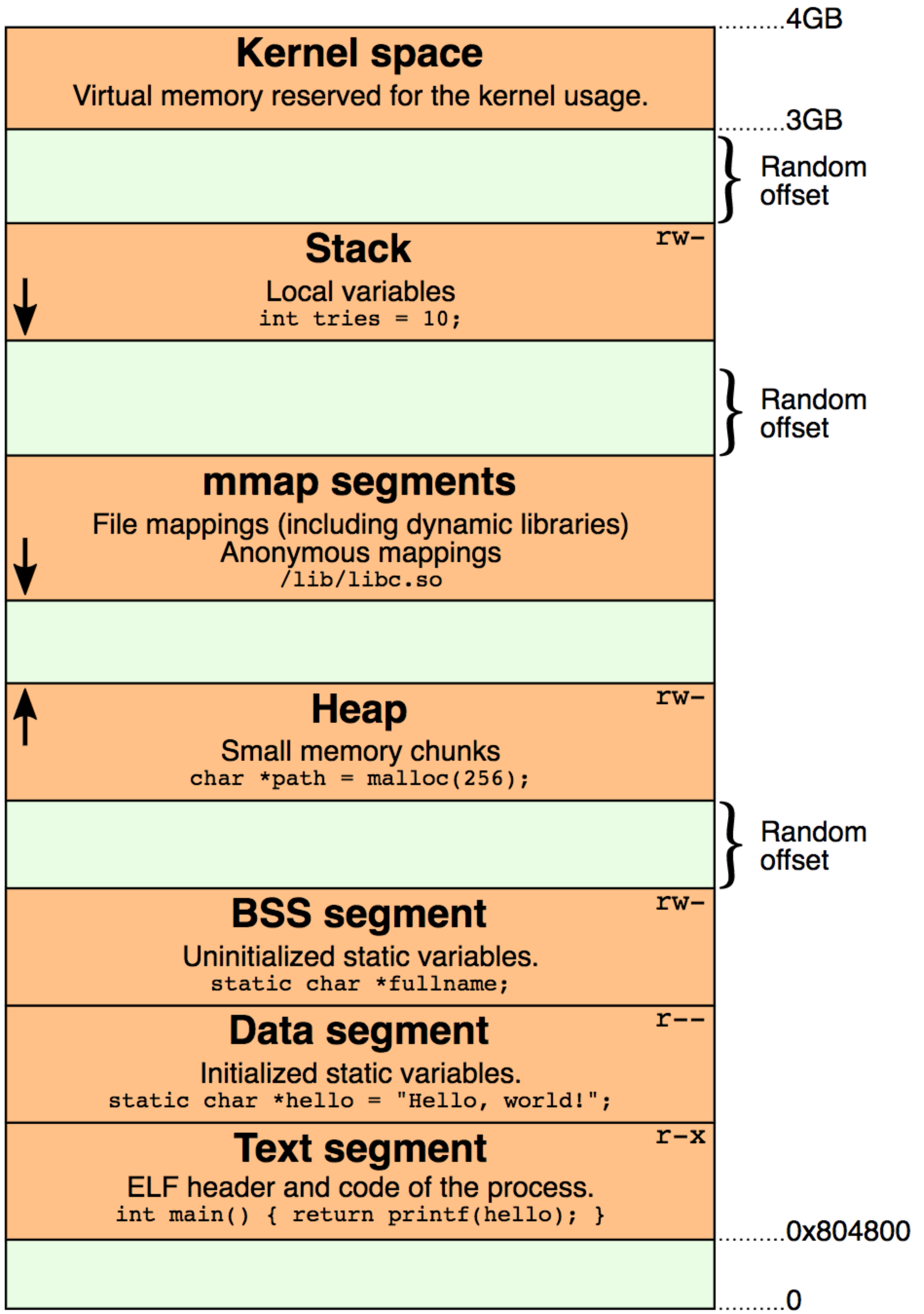
The stack starts growing down! The C++ runtime reserves a portion of memory (that is extended dynamically upon a page fault).

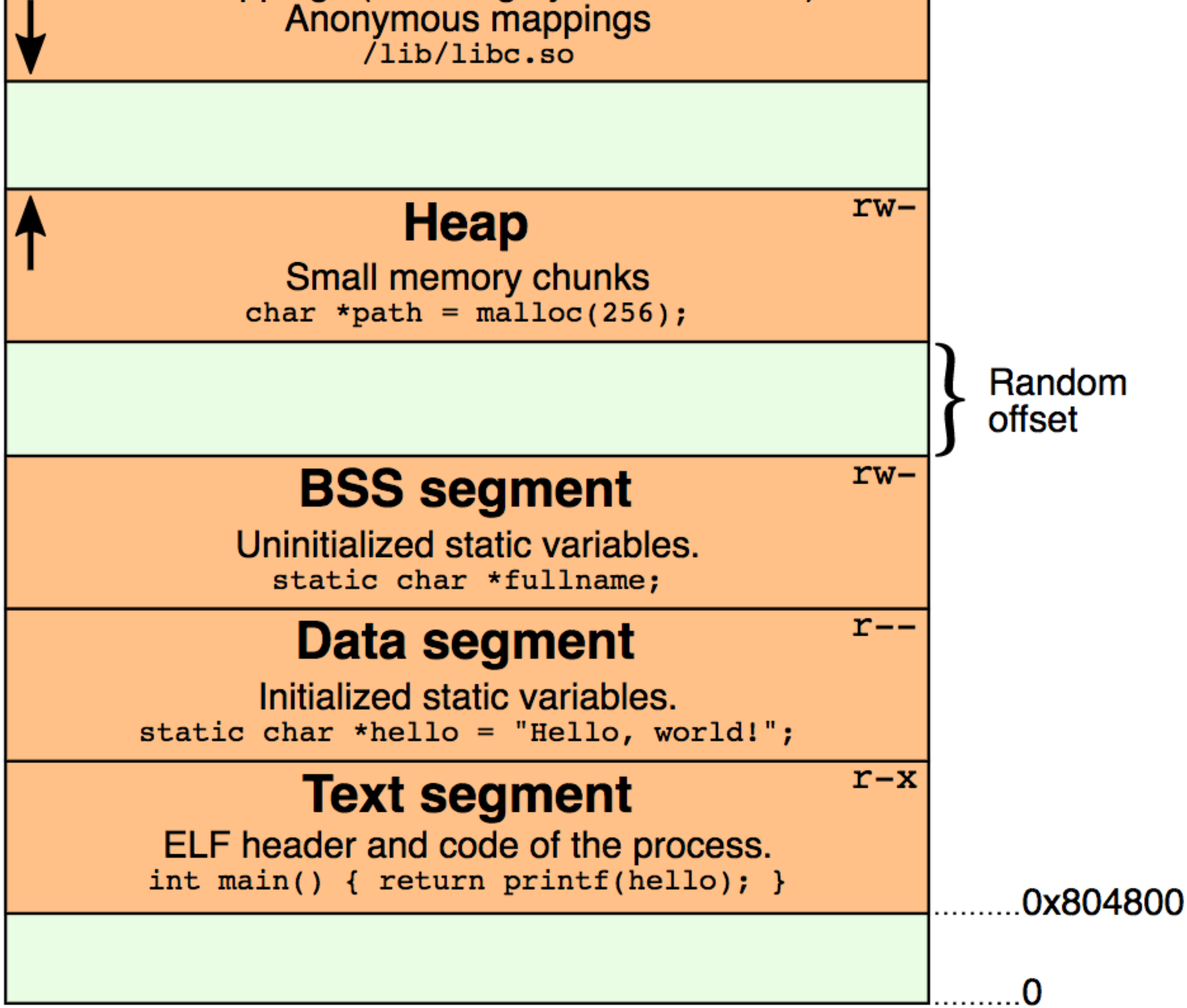
The heap starts below the stack in memory and grows up, page by page.

C++ semantics: memory model

The virtual memory for your C++ binary is organized like so:

Note: The stack grows down. The heap grows up (and is managed by a memory allocator such as malloc in libc).





Anonymous mappings
/lib/libc.so
Heap
rw-
Small memory chunks
char *path = malloc(256);
Random offset
BSS segment
rw-
Uninitialized static variables.
static char *fullname;
Data segment
r--
Initialized static variables.
static char *hello = "Hello, world!";
Text segment
r-x
ELF header and code of the process.
int main() { return printf(hello); }
0x804800
0

C++ semantics: pointers and references

Prefix `*` operation turns a pointer into a reference! `*x` references the value at address `x`.

```
int* x = f(); // x is a pointer to an int
int y = *x;   // *x dereferences the ptr
```

```
int x = f(); // x is an int
int* y = &x; // &x takes address of x
```

Prefix `&` operation turns a reference into a pointer! `&x` is the address of the value referenced by `x`.

C++ semantics: field access, . and ->

```
A& a = f();     // a is a reference to an object
//A a = f();    // same thing
int y = a.y;    // a.y accesses field y of a
```

The . operation restricts a reference to a specific field; here, `a.y` turns a reference to a an object into a reference to its `y` field.

The -> operation dereferences a pointer and accesses a specific field all at once.

```
A* a = f();       // a is a pointer to an object
int y = a->y;     // a->y accesses field y off a
```

C++ semantics: indexing and dereference

Postfix [..] operation turns a pointer into a reference to the element specified as the index

```
int* x = f(); // x is a pointer to an int
int y = x[0]; // x[0] indexes the pointer
```

```
int* x = f(); // x is a pointer to an int
int y = *x;   // this is the same as x[0]
```

If the index is 0, then this is just the same as dereferencing the pointer!

C++ semantics: indexing and dereference

```
int* x = f();     // x is a pointer to an int
int y = *(x+3);   // this is the same as x[3]
```

If the index is non-0, then this is just the same as incrementing the pointer and then dereferencing

This is the same as incrementing the raw address by the appropriate number of bytes. The void* type gives access to the raw address.

```
int* x = f();
// x is a pointer to an int
int y = *(int*)((void*)x
                  + 3*sizeof(int));
// this is ALSO the same as x[3]
```

C++ semantics: Try an example!

```
int arr[8] = {0,5,1,2,3,4,5,9};
int*  x  = arr;         // Derive a ptr from arr
std::cout << arr[1] << std::endl;
// Which value is printed out?
```

C++ semantics: Try an example!

```
int arr[8] = {0,5,1,2,3,4,5,9};
int*  x  = arr;          // Derive a ptr from arr
std::cout << arr[1] << std::endl;
// Which value is printed out?
```

Answer: 5

C++ semantics: Try an example!

```
int arr[8] = {0,5,1,2,3,4,5,9};
int*  x  = arr;          // Derive a ptr from arr
std::cout << &arr << std::endl;
// Which value is printed out?
```

C++ semantics: Try an example!

```
int arr[8] = {0,5,1,2,3,4,5,9};
int*  x  = arr;          // Derive a ptr from arr
std::cout << &arr << std::endl;
// Which value is printed out?
```

Answer: 0xff443120 <— <u>ptr</u> to var `x`

in other words, `**(&arr) == 0`

C++ semantics: Try an example!

```
int arr[8] = {0,5,1,2,3,4,5,9};
int*  x  = arr;           // Derive a ptr from arr
std::cout << (&arr[3])+1 << std::endl;
// Which value is printed out?
```

C++ semantics: Try an example!

```
int arr[8] = {0,5,1,2,3,4,5,9};
int*  x  = arr;          // Derive a ptr from arr
std::cout << (&arr[3])+1 << std::endl;
// Which value is printed out?
```

Answer: 0xecff6604 <— ptr to elem 3

in other words, *((&arr[3])+1) == 3

C++ semantics: Try an example!

```
int arr[8] = {0,5,1,2,3,4,5,9};
int*  x  = arr;
int*  y  = x+3;
int   z  = *y;           // What is z?
int*  a  = &(y[z]);
void* b  = (void*)a + sizeof(int);
int   c  = *((int*)b - 2); // What is c?
```

C++ semantics: Try an example!

```
int arr[8] = {0,5,1,2,3,4,5,9};
int*  x  = arr;
int*  y  = x+3;
int   z  = *y;              // What is z?
int*  a  = &(y[z]);
void* b  = (void*)a + sizeof(int);
int   c  = *((int*)b - 2); // What is c?
```

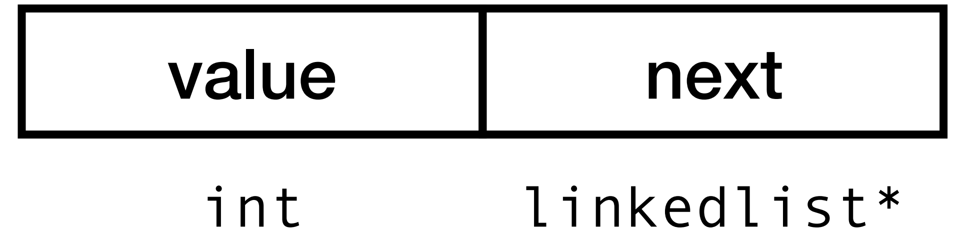
Answer: z == 2 && c == 3

reverse.cpp solution

```
struct linkedlist
{
    int value;
    linkedlist* next;
};

int main()
{
    linkedlist* node = 0; //root
    int n;
    while (std::cin >> n)
    {
        linkedlist* next = node;
        node = new linkedlist();
        node->value = n;
        node->next = next;
    }//…
```

data layout in memory



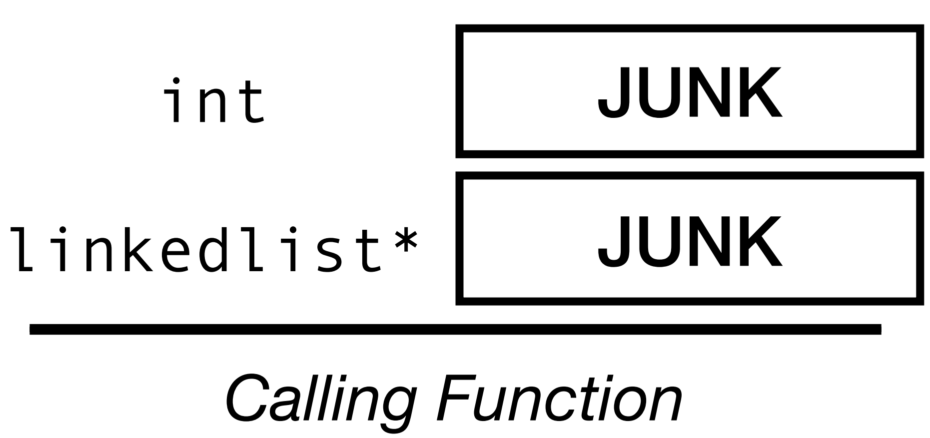
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```

Calling Function

How C++ sees the **stack**

How C++ sees the **heap**

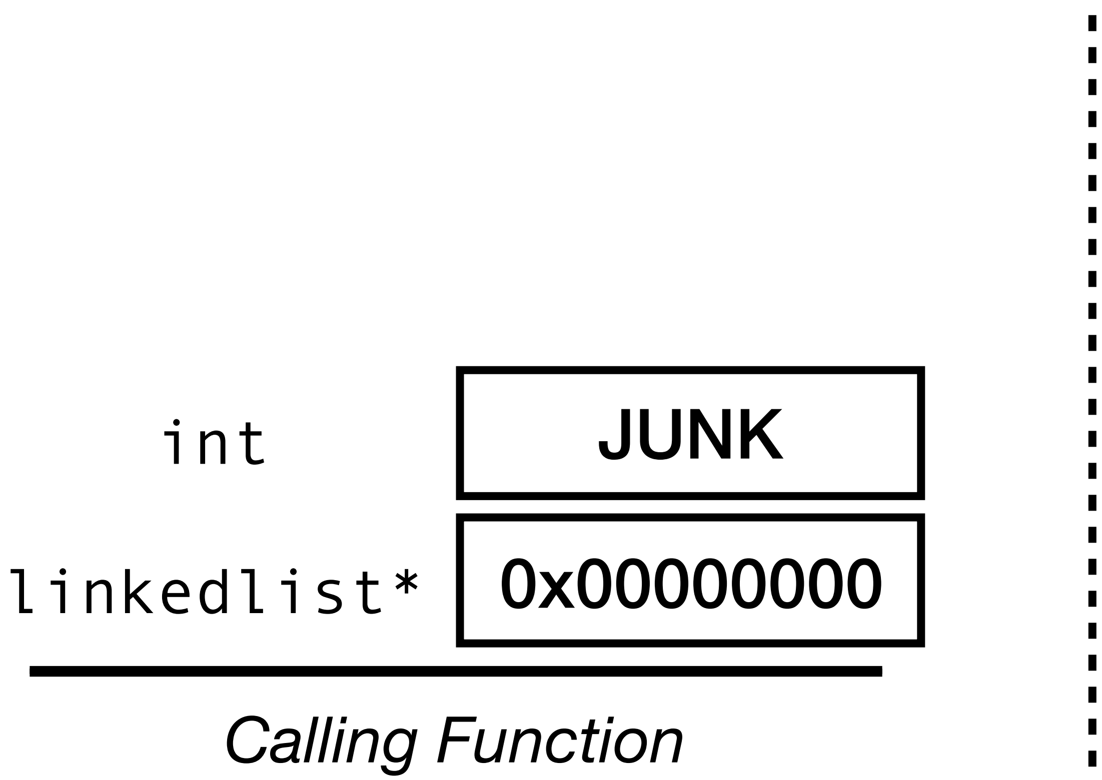
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



How C++ sees the **stack**

How C++ sees the **heap**

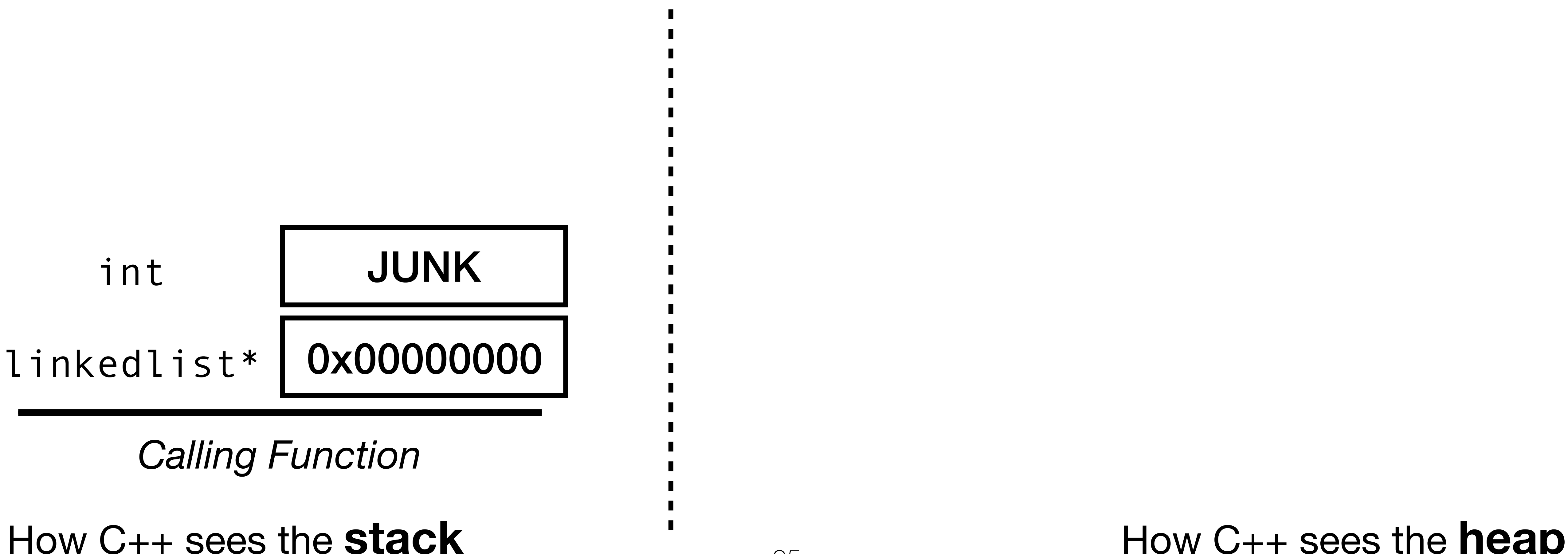
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



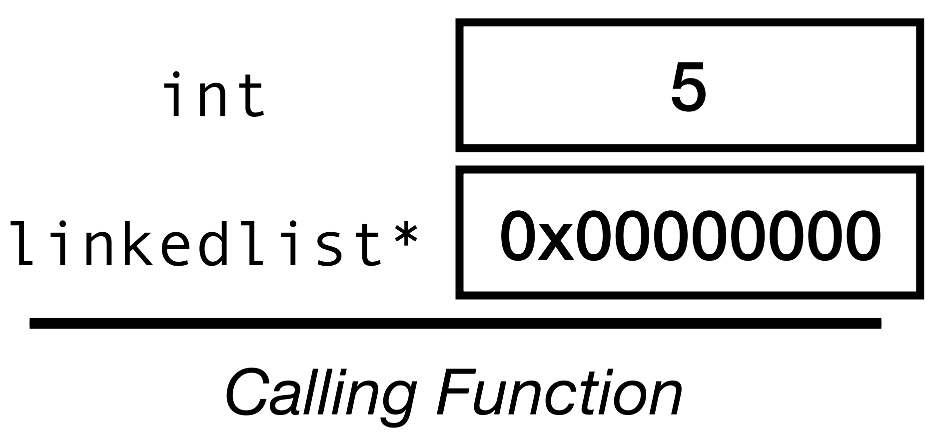
How C++ sees the **stack**

How C++ sees the **heap**

```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



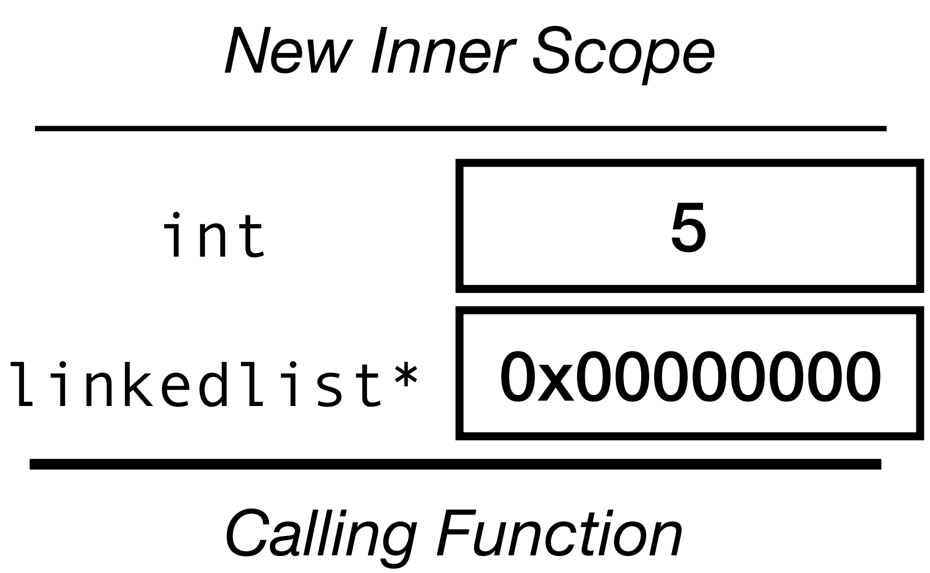
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



How C++ sees the **stack**

How C++ sees the **heap**

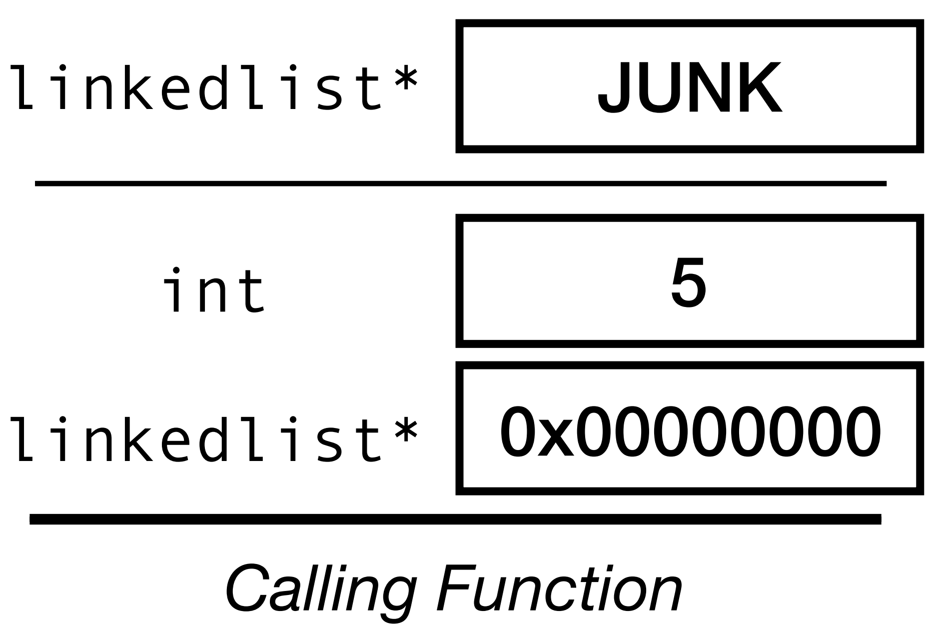
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



How C++ sees the **stack**

How C++ sees the **heap**

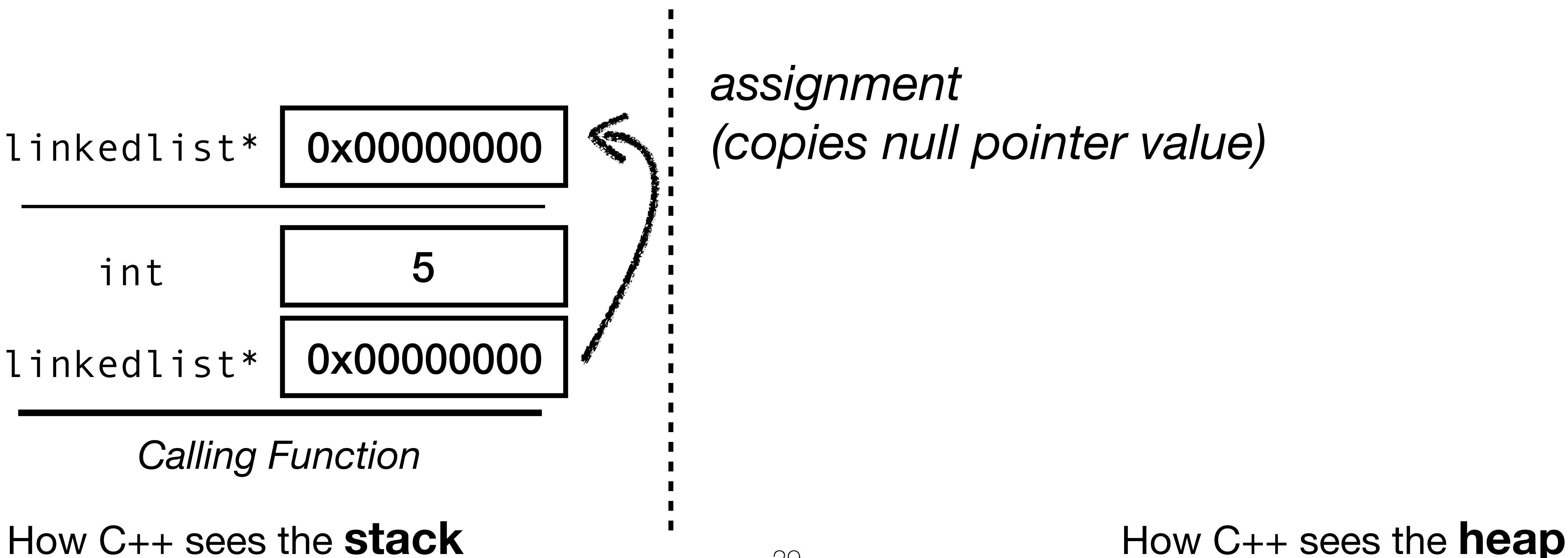
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



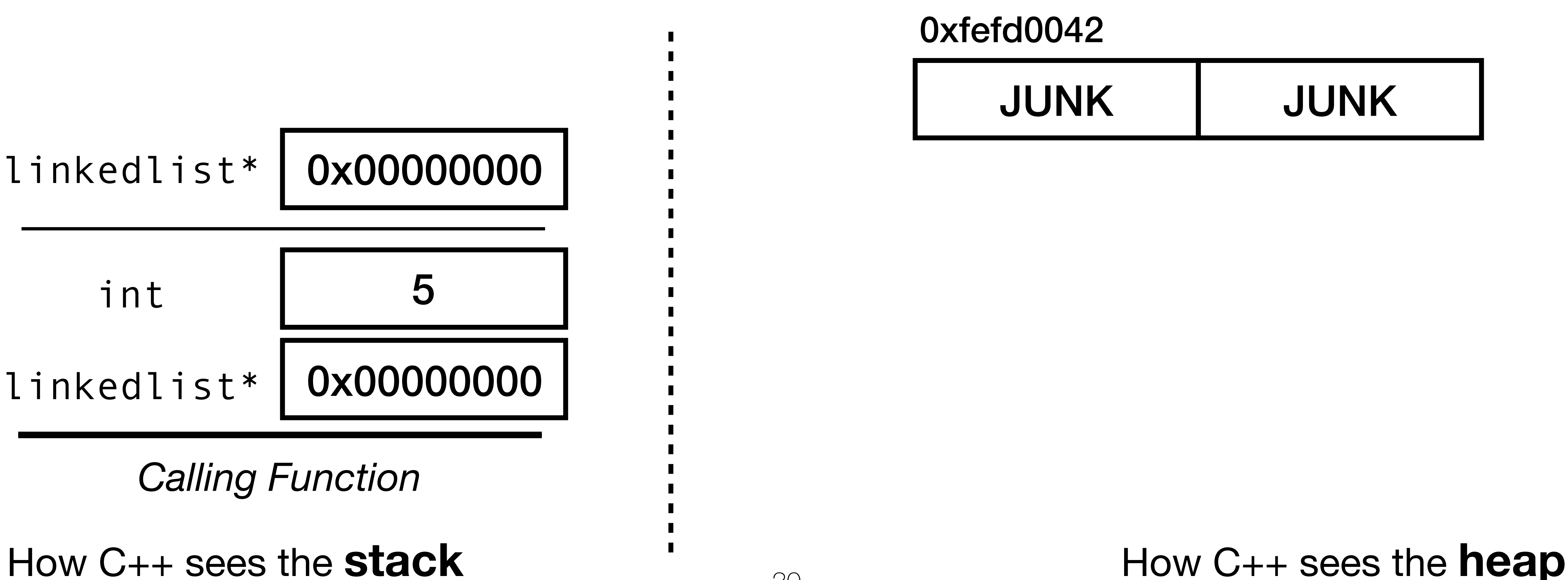
How C++ sees the **stack**

How C++ sees the **heap**

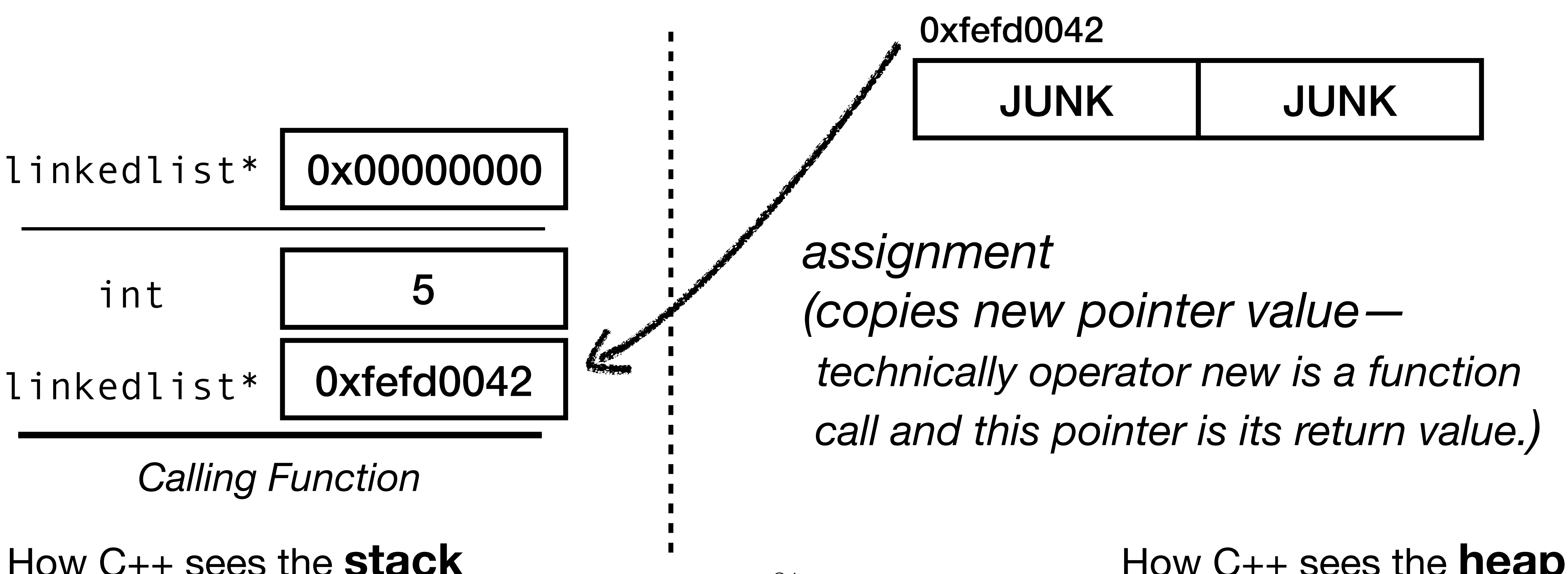
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



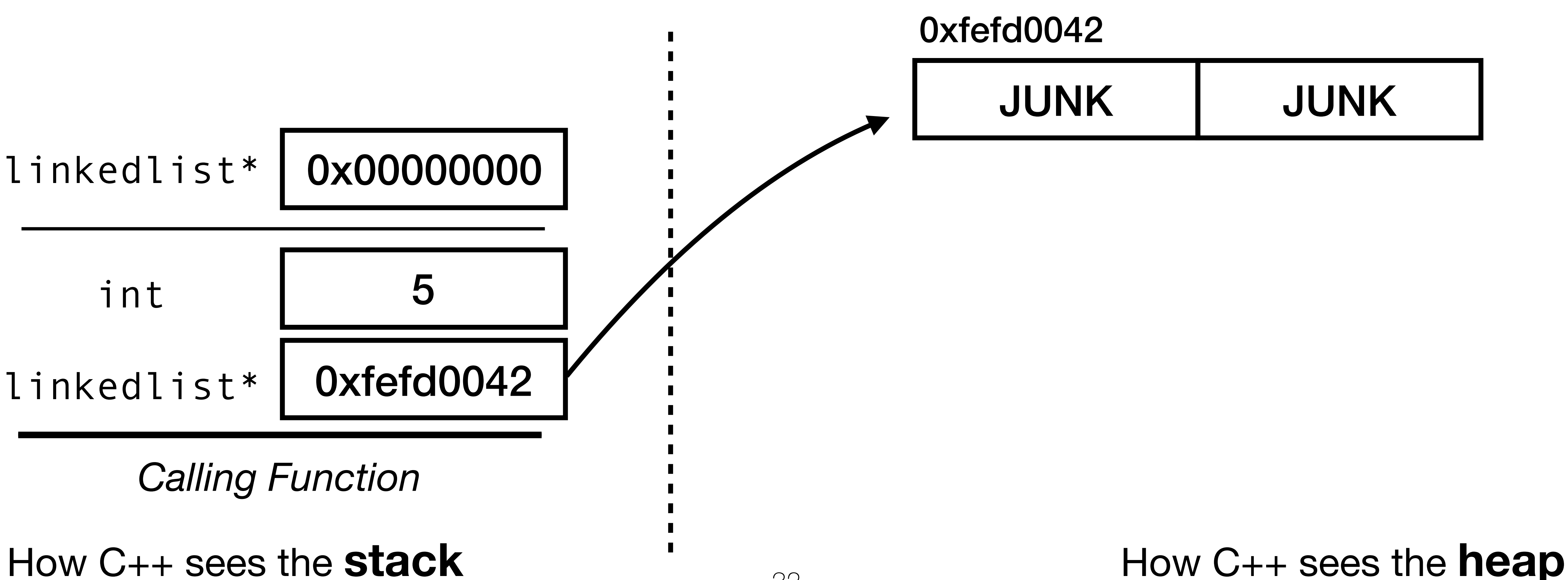
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



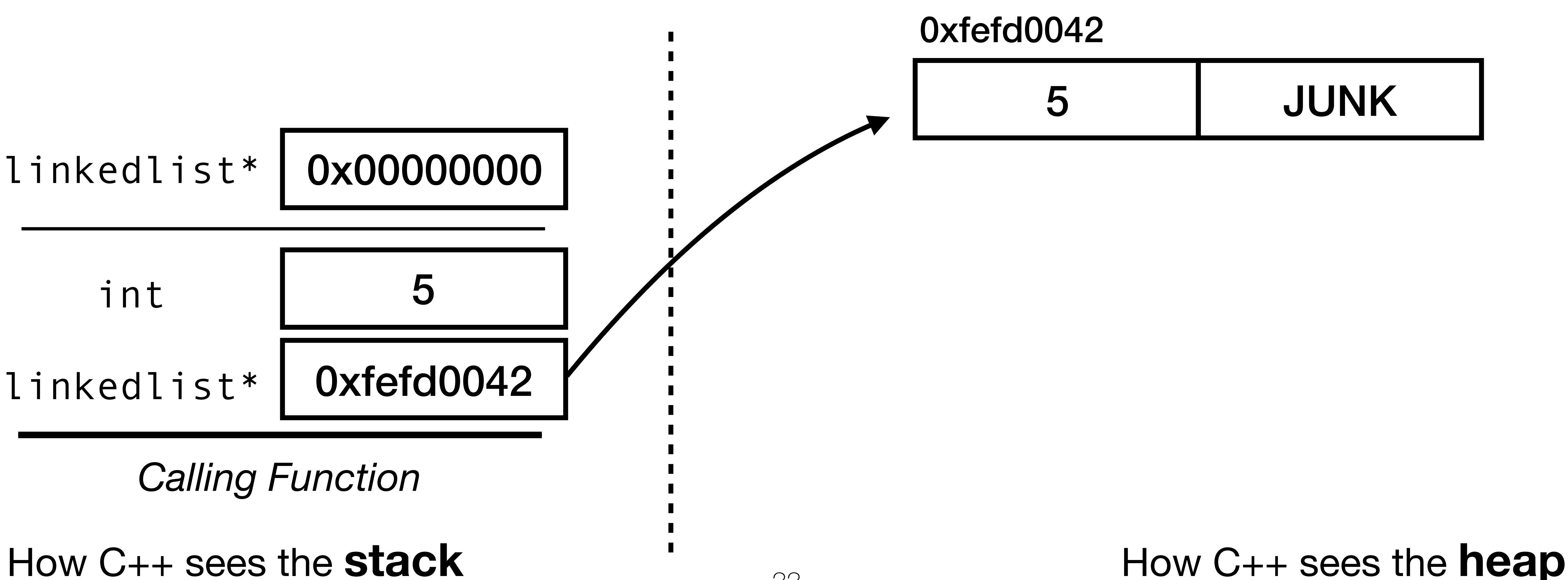
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



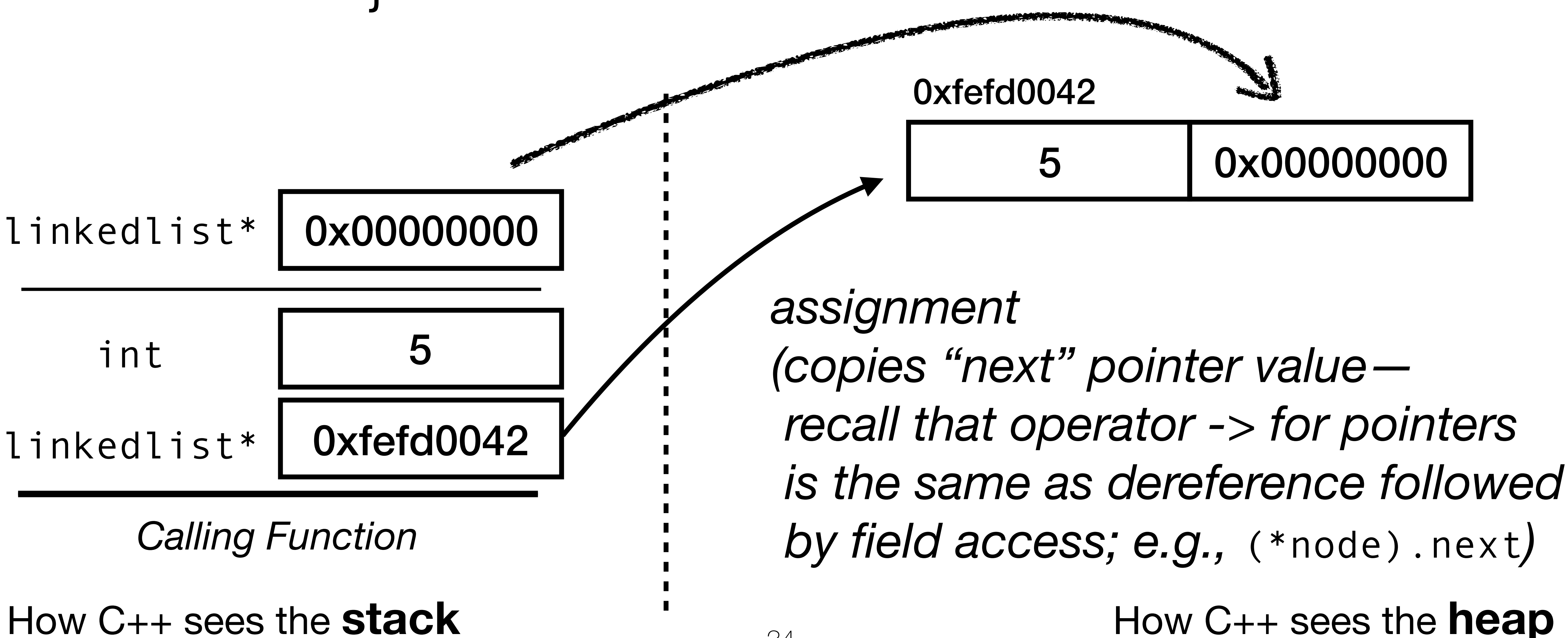
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



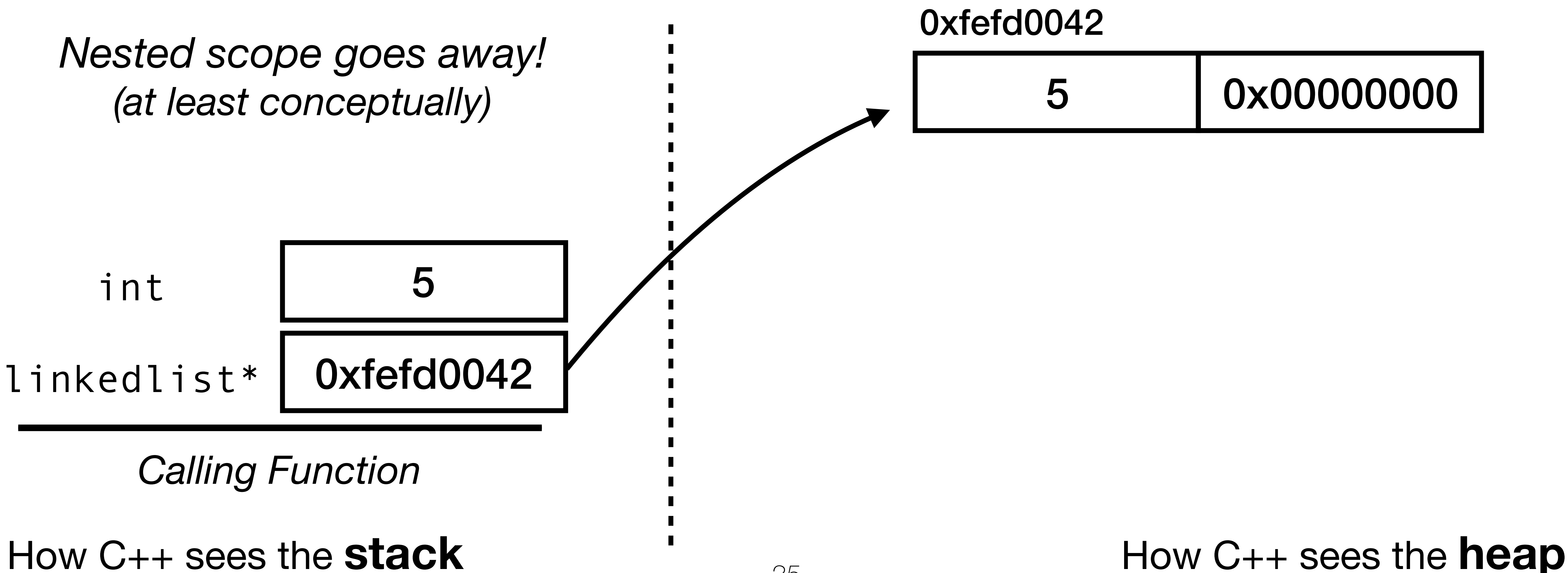
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```

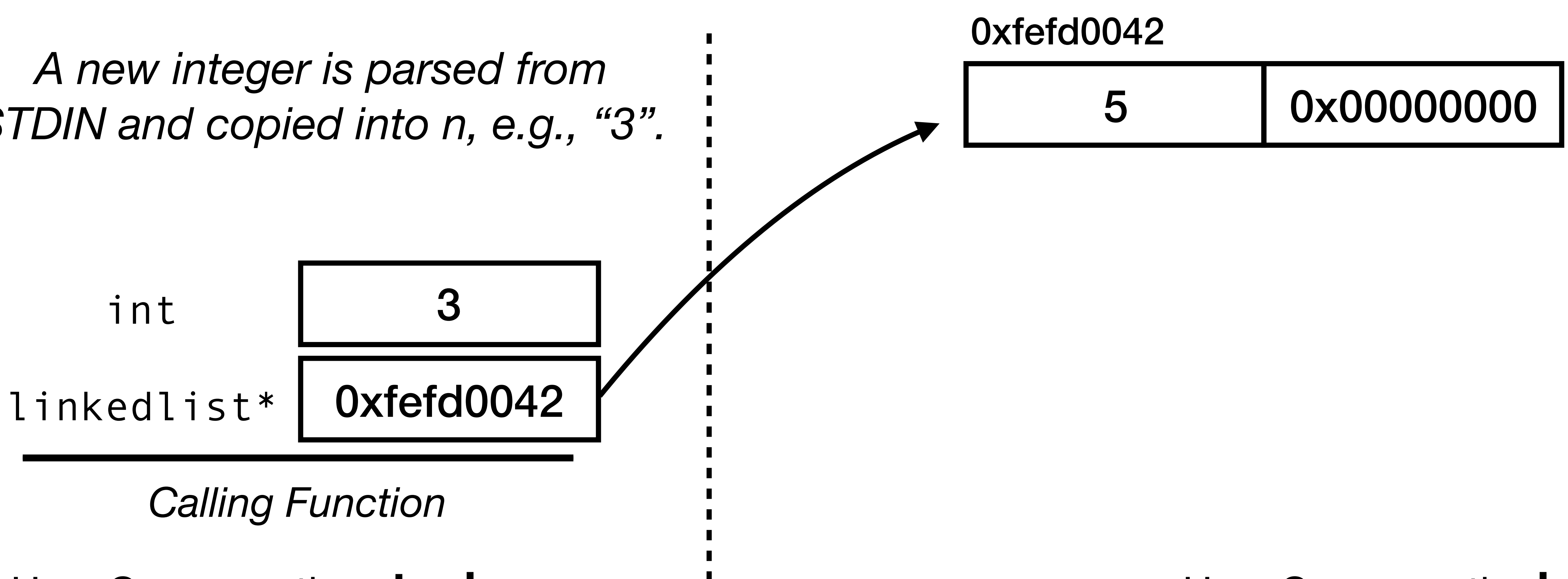


```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```

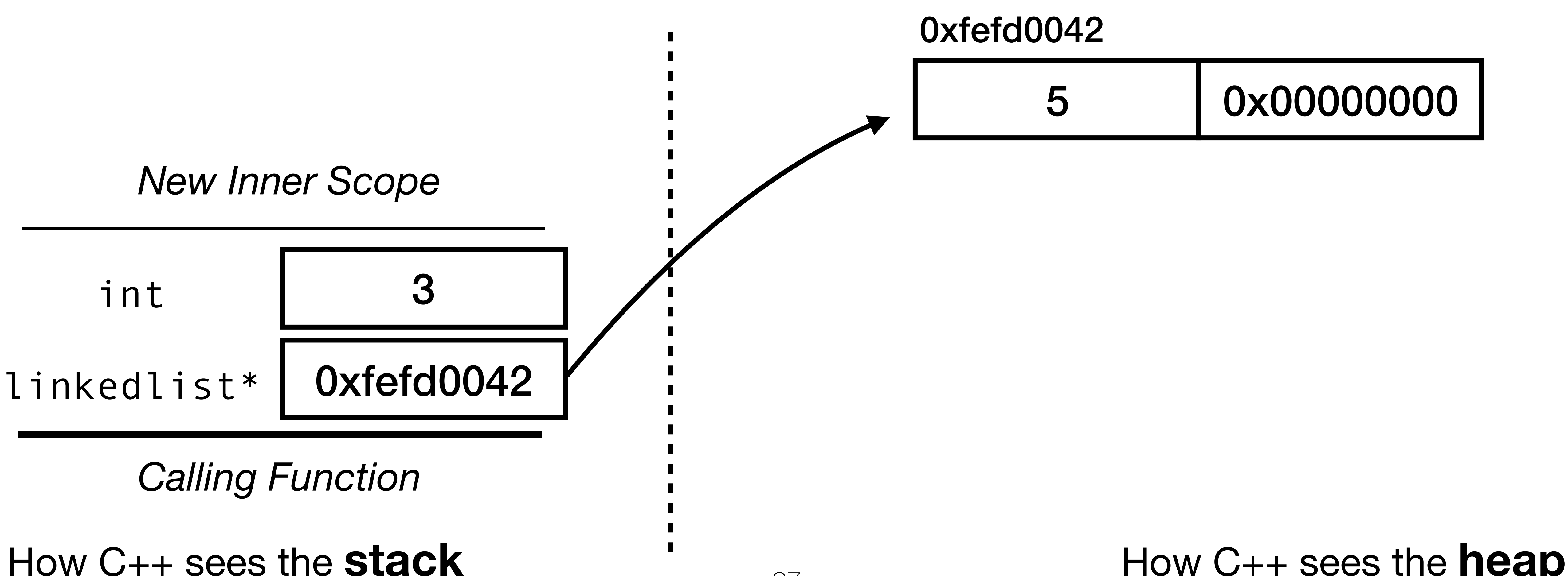
A new integer is parsed from STDIN and copied into n, e.g., “3”.



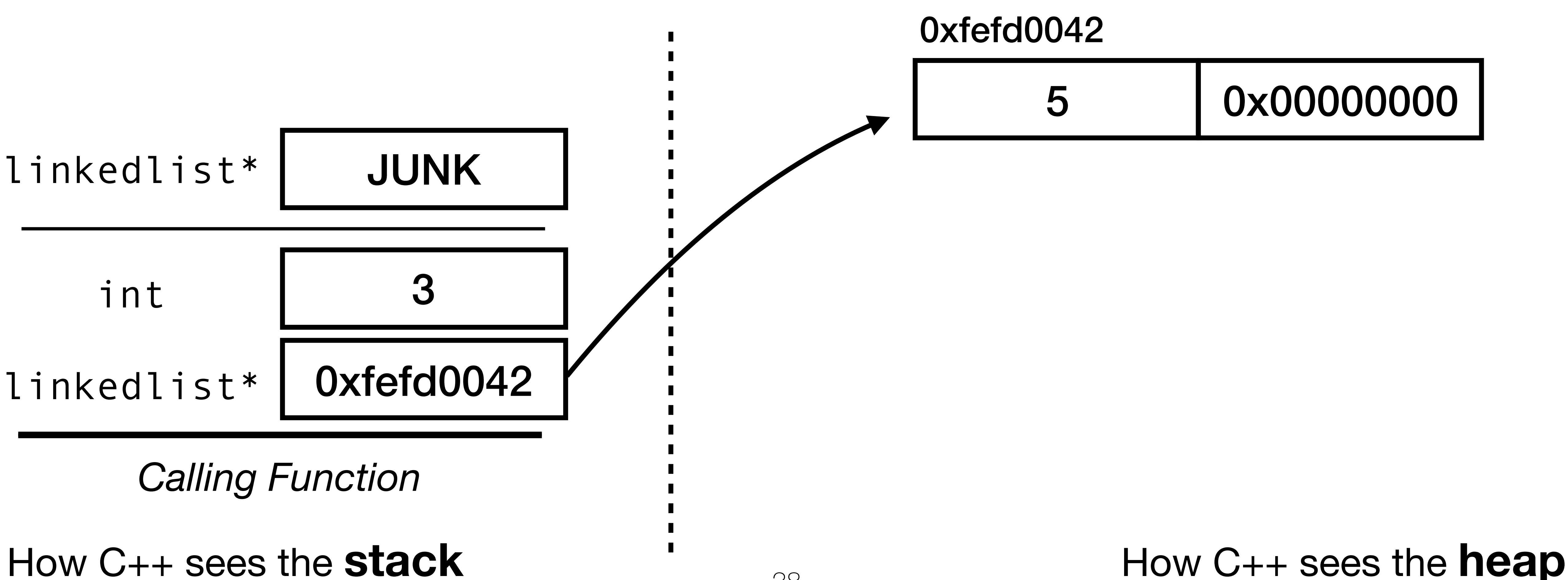
How C++ sees the **stack**

How C++ sees the **heap**

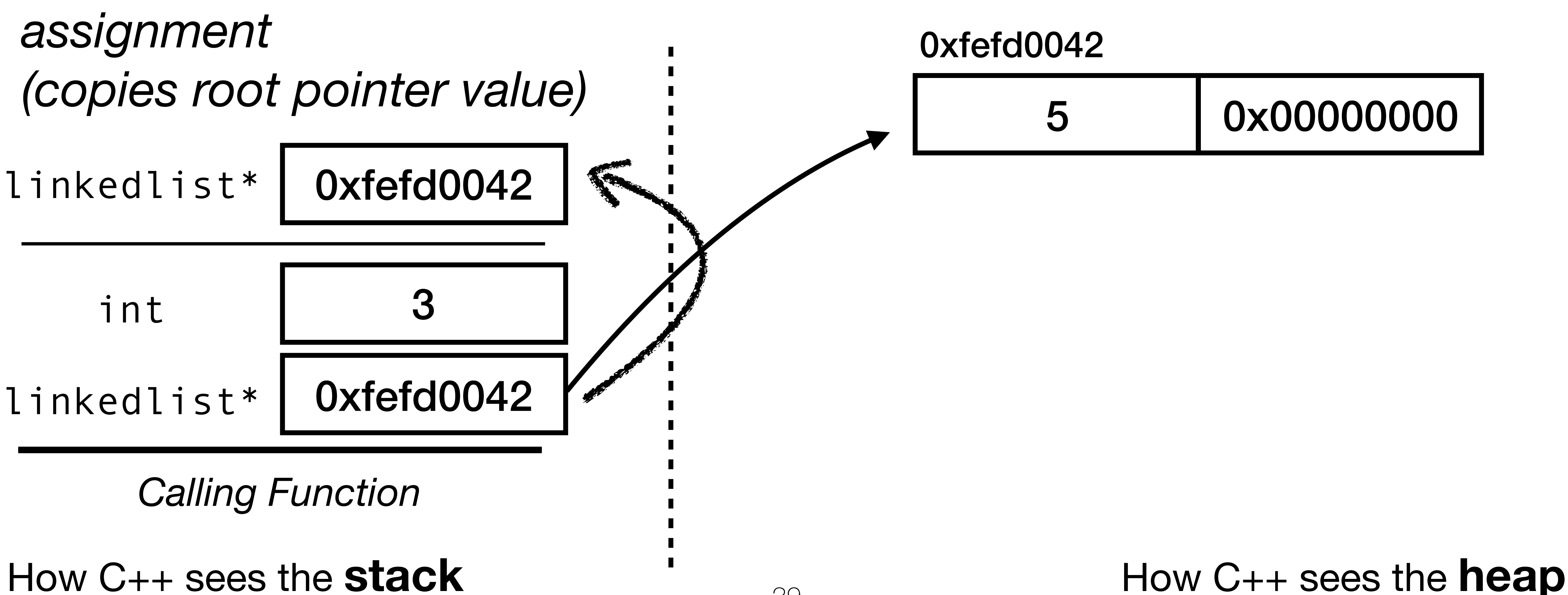
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



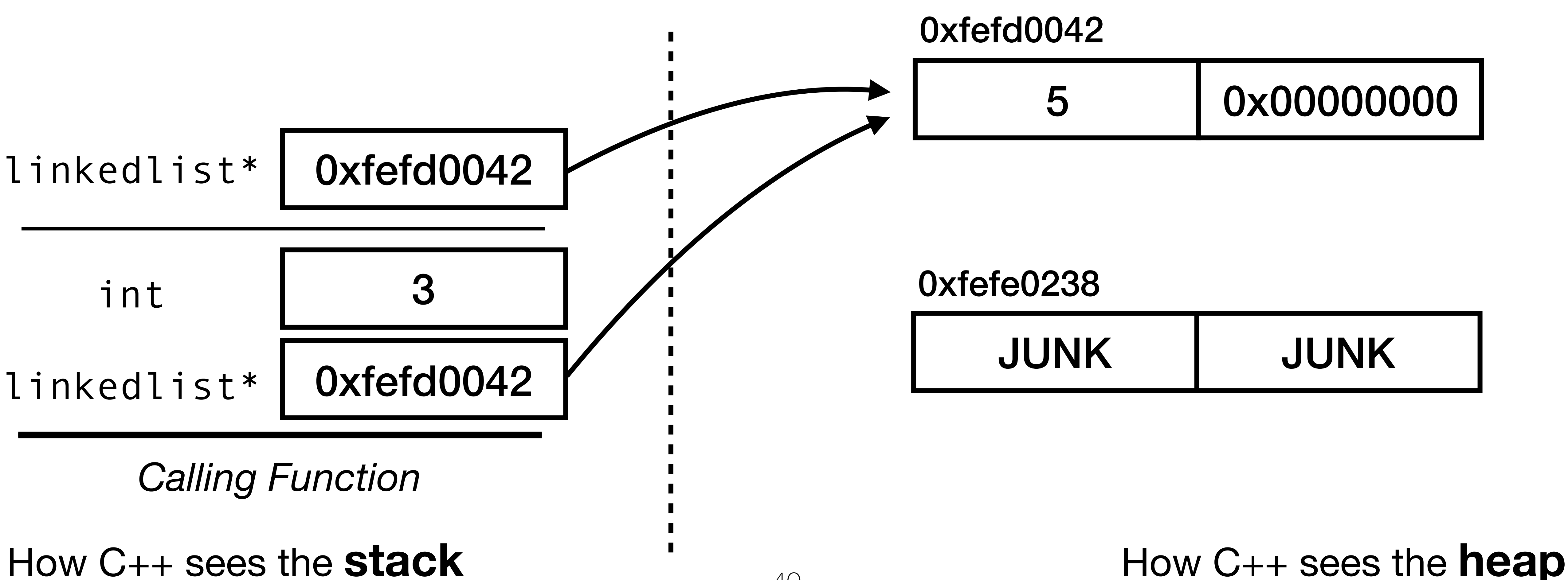
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



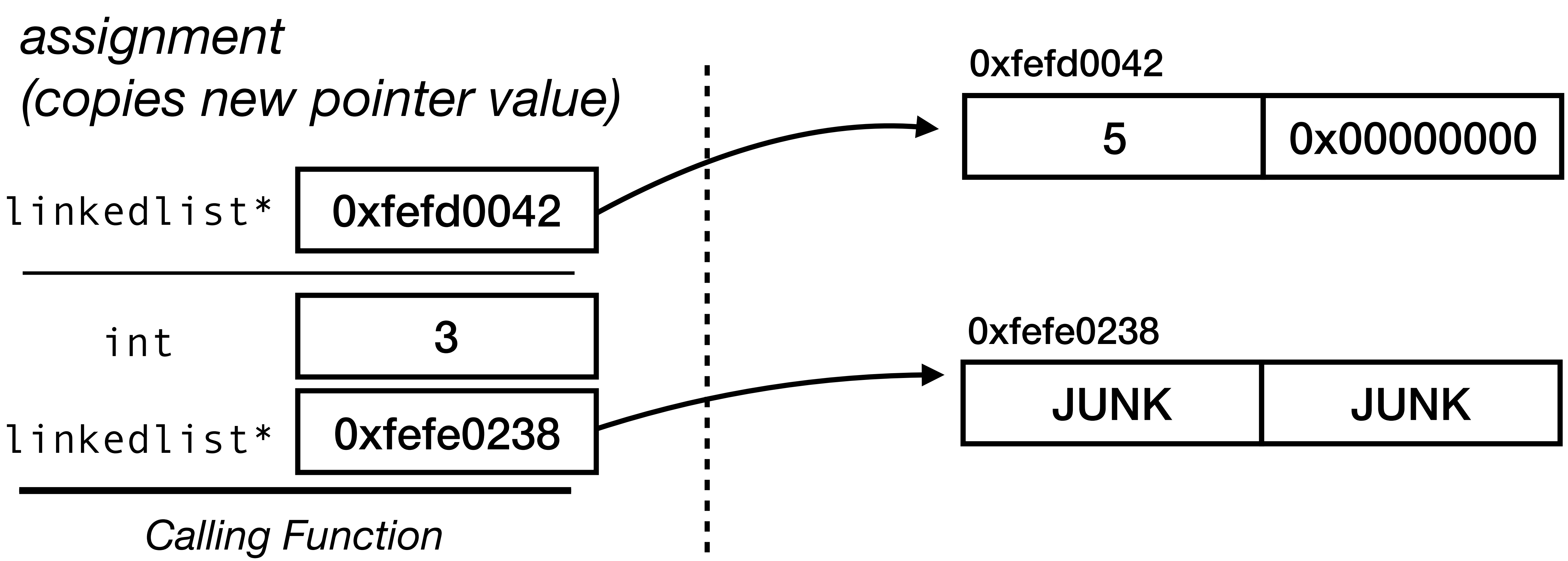
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



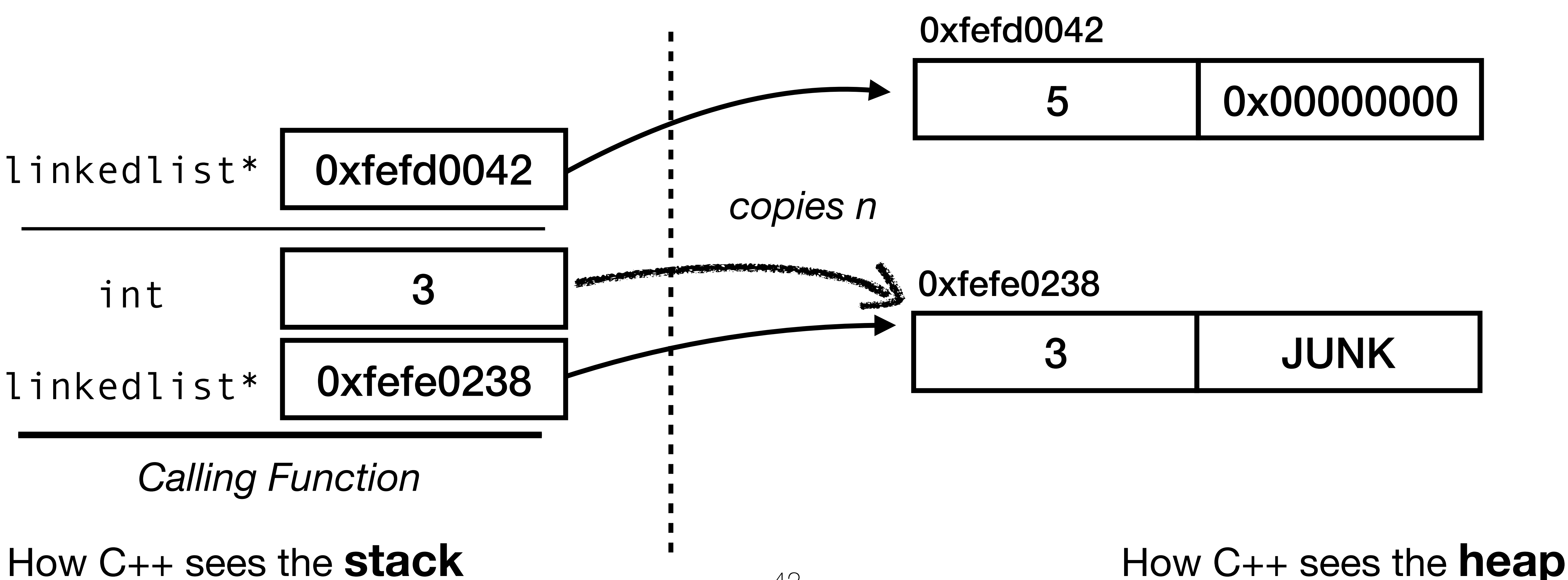
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



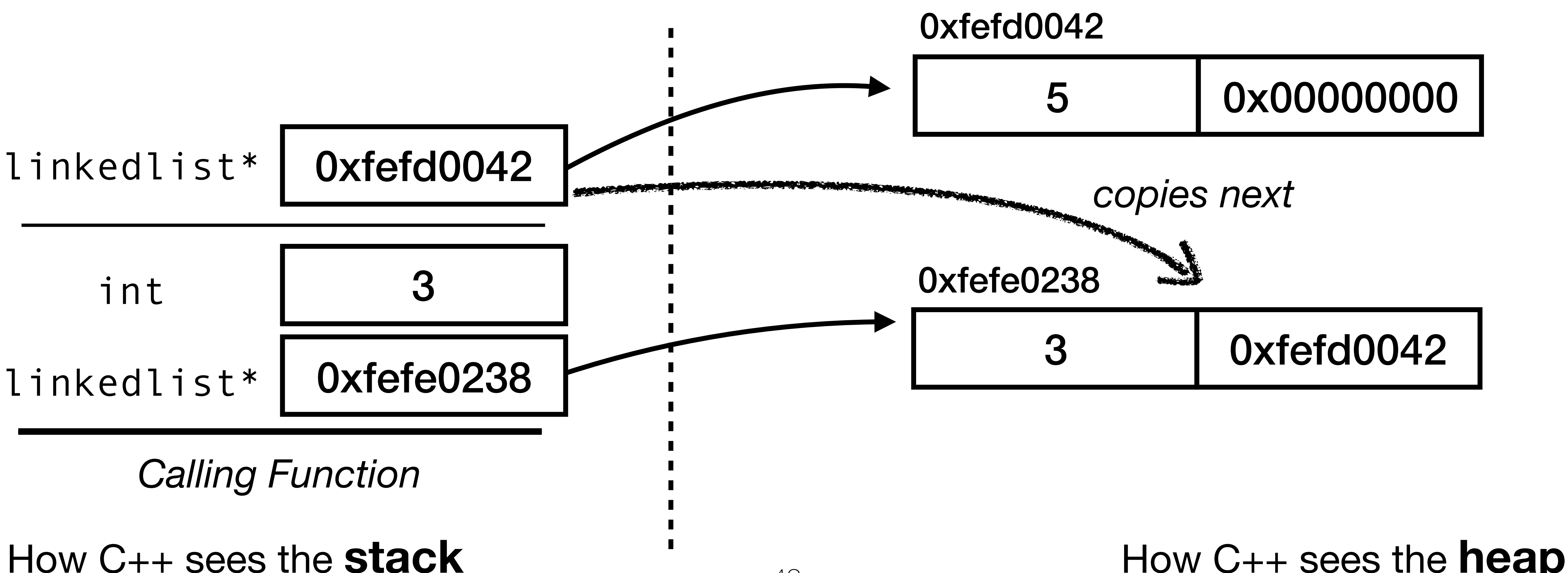
How C++ sees the **stack**

How C++ sees the **heap**

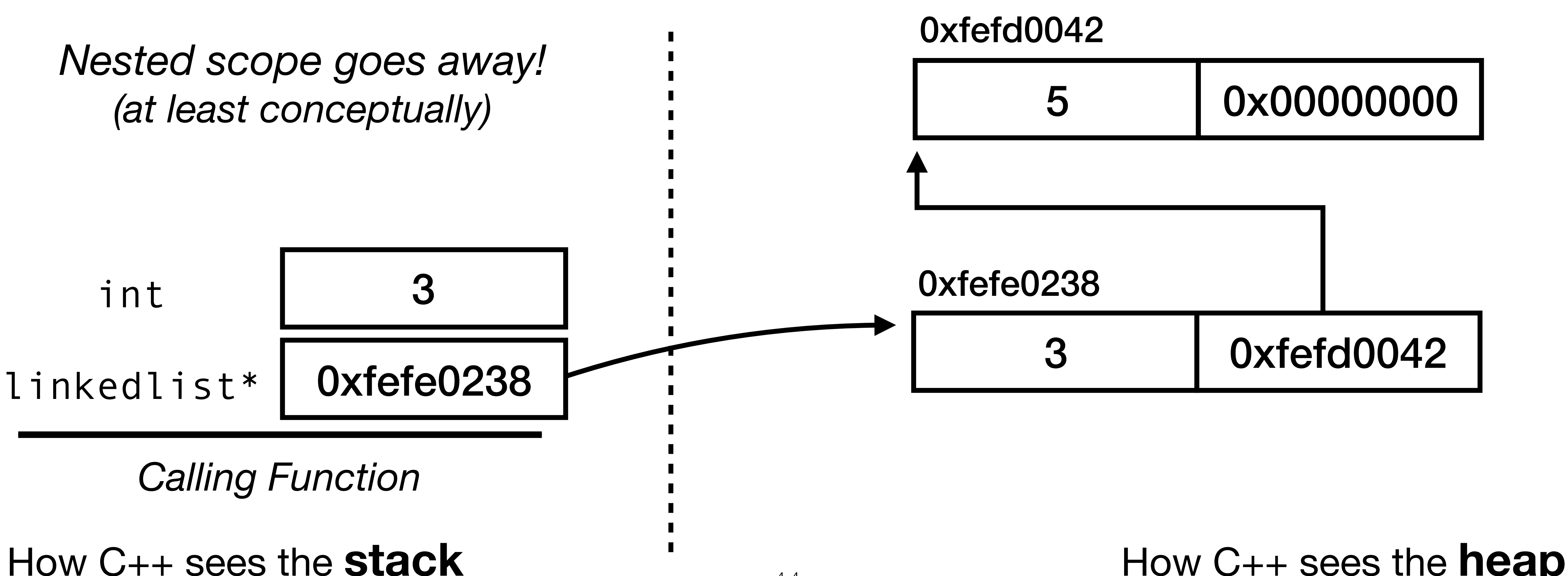
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



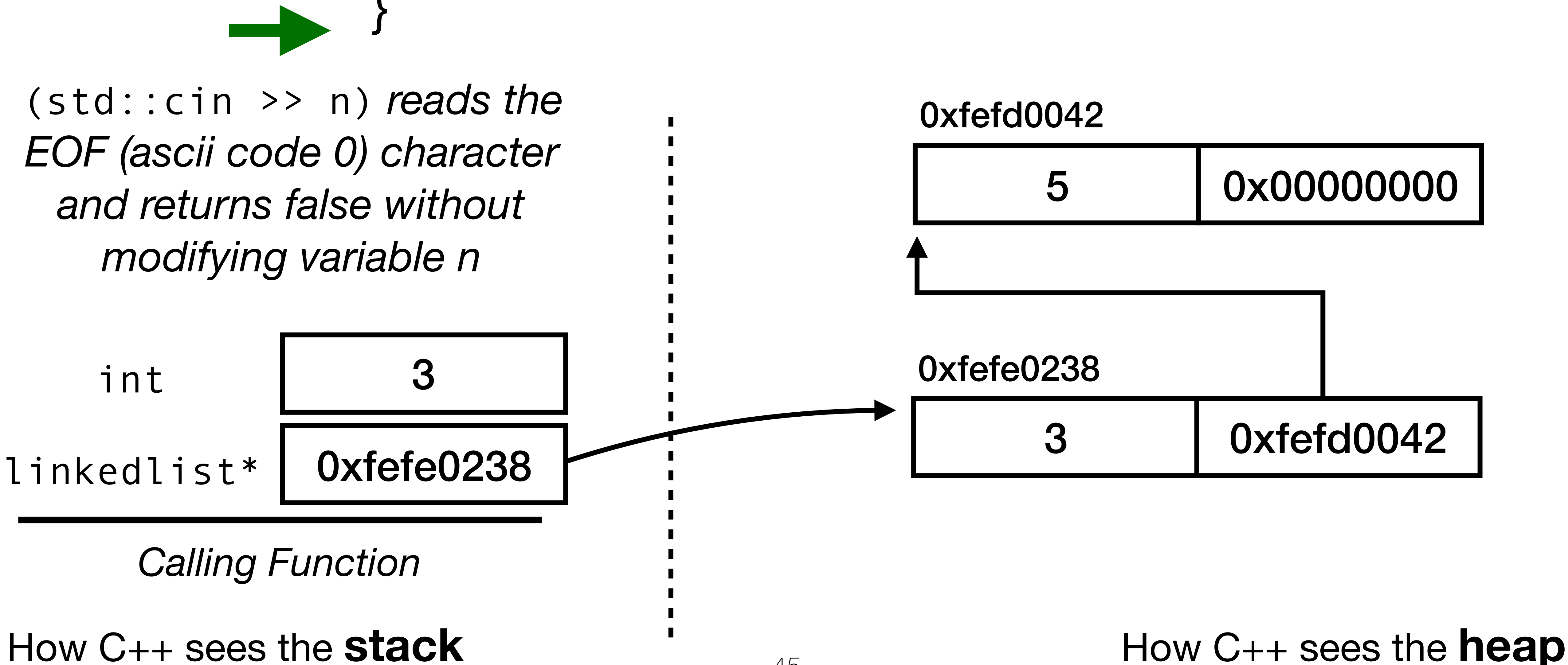
```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



```
linkedlist* node = 0; //root
int n;
while (std::cin >> n)
{
    linkedlist* next = node;
    node = new linkedlist();
    node->value = n;
    node->next = next;
}
```



C++ semantics: taking pointers of stack values

```
char* badalloc()
{
    char bytes[4096] = {0};
    return &bytes[0];
}

int main()
{
    char* arr = badalloc();
    arr[0] = ‘h’;
    arr[1] = ‘i’;
    std::cout << arr << std::endl;
    return 0;
}
```

C++ semantics: Try an example

```
char* badalloc()
{
    char bytes[4096] = {0};
    return &bytes[0];
}

int main()
{
    char* arr = badalloc();
    arr[0] = ‘h’;
    arr[1] = ‘i’;
    std::cout << arr << std::endl;
    return 0;
}
```

What could go wrong when allocating memory this way?

C++ semantics: taking pointers of stack values

```
$ clang++ -o bin badalloc.cpp
badalloc.cpp:8:13: warning: address
of stack memory associated with local
variable 'bytes' returned [-Wreturn-
stack-address]
    return &bytes[0];
            ^~~~~
1 warning generated.
$ ./bin
hi
$
```

C++ semantics: taking pointers of stack values

```
char* passthrough(char* ptr)
{
    return ptr;
}

char* badalloc()
{
    char bytes[4096] {0};
    return passthrough(&bytes[0]);
}

int main()
{
    …
```

C++ semantics: taking pointers of stack values

```
$ clang++ -o bin badalloc.cpp
$ ./bin
hi
$
```

The compiler wont always
catch this problem for us!

C++ semantics: taking pointers of stack values

Now we can try making the buffer small!

```
char* badalloc()
{
    char bytes[8] = {0};
    return passthrough(&bytes[0]);
}

int main()
{
    char* arr = badalloc();
    arr[0] = ‘h’;
    arr[1] = ‘i’;
    std::cout << arr << std::endl;
    return 0;
}
```

C++ semantics: taking pointers of stack values

```
$ clang++ -o bin badalloc.cpp
$ ./bin
\300I\211\350\376^?
$
```

Now the call to `std::cout` itself tramples on this stack space and overwrites these bytes with values that are, to us, junk!

Quiz

If class Foo takes up 64 bytes on the stack, how much memory will be used in the following code:

```
Foo *f = new Foo(…);
int  x = (*f).value;
```

- 64 (Foo) + 8 (pointer to Foo) + 4 (int)?
- 64 (Foo) + 8 (pointer to Foo) + 64 (deref pointer) + 4 (int)?

Quiz

If class Foo takes up 64 bytes on the stack, how much memory will be used in the following code:

```
Foo *f = new Foo(…);
int  x = (*f).value;
```

- **64 (Foo) + 8 (pointer to Foo) + 4 (int)?**
- 64 (Foo) + 8 (pointer to Foo) + 64 (deref pointer) + 4 (int)?

Since *f gives back a *reference*, no additional copying is done

Quiz

How many times is a Foo constructor called?

```
void m(Foo v) { … }
Foo f(…)
int  x = m(f);
```

Quiz

How many times is a Foo constructor called?

```
void m(Foo v) { … }
Foo f(…)
int  x = m(f);
```

- **Twice! Once for Foo, once for the copy constructor**

Quiz

How many times is a Foo constructor called?

```
void m(Foo &v) { … }
Foo f(…)
int  x = m(f);
```

Quiz

How many times is a Foo constructor called?

```
void m(Foo &v) { … }
Foo f(…)
int  x = m(f);
```

- **Once! Second is passed by *reference***

Quiz

If copying by reference is faster, why not just *always* pass by reference?

Quiz

If copying by reference is faster, why not just *always* pass by reference?

Passing by reference might change the value to the caller. Caller needs to know what might happen. Const reference guarantees no change. Prefer const ref.

Quiz

Why not just always pass by pointer?

Quiz

Why not just always pass by pointer?

Basically: raw pointers are dangerous. It's easy to mess them up. Use references when possible, since they are a "less powerful" datatype

How Objects Work

C++ dynamic dispatch: Try an example!

```
class B
{
    virtual int f() { return 1; }
};
class A : public B
{
    virtual int f() { return 2; }
};


B* a = new A(); // Get a pointer to an A obj
std::cout << a->f() << std::endl;
```

// Which value is printed out?

C++ dynamic dispatch: Try an example!

```
class B
{
    virtual int f() { return 1; }
};
class A : public B
{
    virtual int f() { return 2; }
};


B* a = new A(); // Get a pointer to an A obj
std::cout << a->f() << std::endl;
```

// Which value is printed out? ANSWER: 2

Function pointers

```
int add1(int x) { return x+1; }
```

In stored-program machines, all code sits somewhere in memory.

In C/C++, you can obtain pointers to functions at run-time, and invoke them! The pointer for `add1` can be obtained with:

```
&add1
```

```c
int add1(int x) { return x+1; }


int main()
{
    int (*f)(int) = &add1;

    // …

    int four = (*f)(3);
}
```

A function pointer, `cmp`, passed to `sort` as an argument.

```
int sort(int* x, int len, bool (*cmp)(int,int))
{
    // …

        // …
            if ((*cmp)(*x,*y))
            {
                swap(*x,*y);

                // …
            }

    // …
}
```

The function pointer, `cmp`, dereferenced and invoked.

```
{
    // …

    sort(buff, length, &lessthan);

    // …
}
```

A pointer to function `lessthan` **is passed into** `sort`**.**

C++: Try an example!

Talk to your neighbors. Can you think of another way to parameterize a `sort` method over the comparison predicate to be used?

A function pointer, `cmp`, type `int x int -> bool`, is a template parameter to `sort`.

```
template <bool (*cmp)(int,int)>
int sort(int* x, int len)
{
        // …
            if ((*cmp)(*x,*y))
            {
                swap(*x,*y);
                // …
```

Templated function `sort` is invoked with a template parameter like so: `sort<…>(…)`

```
int main()
{
    // …
    sort<&lessthan>(buff, length);
```

C++ dynamic dispatch: class polymorphism

```
class Cmp
{
    virtual bool cmp(int x, int y) = 0;
};
class LessThan : public Cmp
{
    virtual bool cmp(int x, int y)
    { return x < y; }
};
class GreaterThan : public Cmp
{
    virtual bool cmp(int x, int y)
    { return x > y; }
};
```

An instance of type `Cmp`, `cmp`, has overloaded method `cmp`.

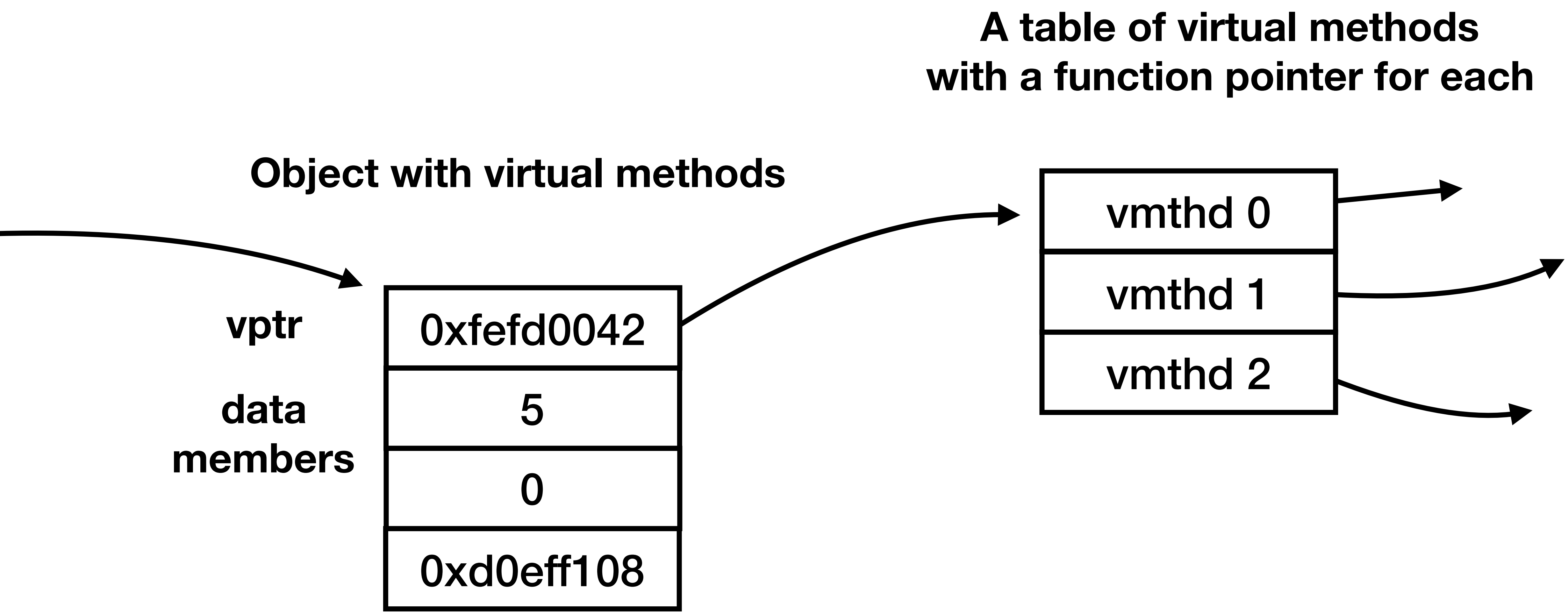
```
int sort(int* x, int len, const Cmp& cmp)
{
        // …
            if (cmp.cmp(*x,*y))
            {
                swap(*x,*y);
                // …


int main()
{
    // …
    LessThan lessthan;
    sort(buff, length, lessthan);
```

Pass in object `lessthan` by reference to polymorphic type Cmp supporting the Cmp::cmp(int, int) member.

Virtual Tables (vtables)

Virtual Tables (vtables)



```
class Animal
{
    virtual const char* name() = 0;
    virtual int weight() const = 0;
    virtual void eat(Animal* prey)
    {
        if (this->weight()
                < 2 * prey->weight())
            return;
        delete prey;
        std::cout << prey->name()
                  << " was eaten!\n";
    }
};
```

```
class Mouse : public Animal
{
    int grams;

    Mouse(int grams)
        : grams(grams) {}

    virtual const char* name()
    {
        return “Mouse”;
    }

    virtual int weight() const
    {
        return this->grams;
    }
};
```

```
class Cat : public Animal
{
    Cat() {}

    virtual const char* name()
    {
        return “Cat”;
    }

    virtual int weight() const
    {
        return 4260;
    }
};
```

```
class Giraffe : public Animal
{
    virtual const char* name()
    {
        return “Giraffe”;
    }
    virtual int weight() const
    {
        return 1570000;
    }
    virtual void eat(Animal* prey)
    {
        std::cout << this->name()
                  << “ wont eat that.\n”;
    }
};
```

```
// vtable struct for Animal subclasses
struct AnimalVTable
{
    const char* (*name)(void*);
    int (*weight)(const void*);
    void (*eat)(void*,void*);

    AnimalVTable(const char* (*name)(void*),
                 int (*weight)(const void*),
                 void (*eat)(void*,void*))
      : name(name), weight(weight), eat(eat)
    {}
};

// Allocate a vtable for each concrete Animal
AnimalVTable mouse_vtable(&nameMouse,
                          &weightMouse,
                          &eatAnimal);
```

```
// Class Mouse compiled to a struct
struct Mouse
{
   AnimalVTable* vptr;
   int grams;
};


// An allocator/constructor for Mouse
Mouse* newMouse(int grams)
{
    Mouse* m = (Mouse*)malloc(sizeof(Mouse));
    m->vptr = &mouse_vtable;
    m->grams = grams;
    return m;
}
```

```
// A name method for Mouse instances
const char* nameMouse(void* _ths)
{
    return “Mouse”;
}


// A weight method for Mouse instances
int weightMouse(const void* _ths)
{
    const Mouse* ths = (const Mouse*)_ths;
    return ths->grams;
}
```

```
// Looks up the vtable for an object
VTable* vtable(void* obj)
{
    return (VTable*)((void**) obj)[0];
}
```

```
{
    // To call a member function f:
    // e.g., obj->f(arg0, arg1, …);

    vtable(obj)->f(obj, arg0, arg1, …);
}
```

```
// Looks up the vtable for an Animal object
AnimalVTable* vtable(void* obj)
{
    return (AnimalVTable*)((void**) obj)[0];
}

// A default eat method for Animals
void eatAnimal(void* ths, void* prey)
{
    if (vtable(ths)->weight(ths)
            < 2 * vtable(prey)->weight(prey))
        return;
    delete prey; // vtable(prey)->~Animal…
    std::cout << vtable(prey)->name(prey)
              << " was eaten!\n";
}
```

Try an example:

How do you define the constructor for Giraffe?

```
// Class Giraffe compiled to a struct
struct Giraffe
{
   AnimalVTable* vptr;
   // No data members
};

AnimalVTable giraffe_vtable(&nameGiraffe,
                            &weightGiraffe,
                            &eatGiraffe);

// An allocator/constructor for Giraffe
Giraffe* newGiraffe()
{
    Giraffe* g = new Giraffe();
    g->vptr = giraffe_vtable;
    return g;
}
```

Try an example:
How do you define the virtual member functions for Giraffe?

```
const char* nameGiraffe(void* _ths)
{
    return "Giraffe";
}

int weightGiraffe(const void* _ths)
{
    return 1570000;
}

void eatGiraffe(void* _ths)
{
    Giraffe* ths = (Giraffe*)_ths;
    std::cout << vtable(ths)->name(ths)
              << " wont eat that.\n";
}
```