

## C++ Objects

David L. Levine  
 Christopher D. Gill  
 Department of Computer Science  
 Washington University, St. Louis  
 levine,cdgill@cs.wustl.edu

<http://classes.cec.wustl.edu/~cs342/>

## C++ Objects

- Code and data
- Memory allocation
- What is an object? How do I create an object?
- Object lifetime
- Where do I instantiate objects? Instantiation Examples
- Objects, references, pointers, arrays
- Header Files
- Things to watch out for

## Code and Data

### DECLARATION

```
class List_Node {
// ...
private:
int item_;
List_Node *next_;
};
```

Class **declaration** does  
 NOT generate executable  
 code! Or, allocate any memory!

### DATA

```
List_Node list_node;
```

Instantiates a List\_Node:  
 creates a List\_Node object.  
 Allocates memory for the object.

### CODE

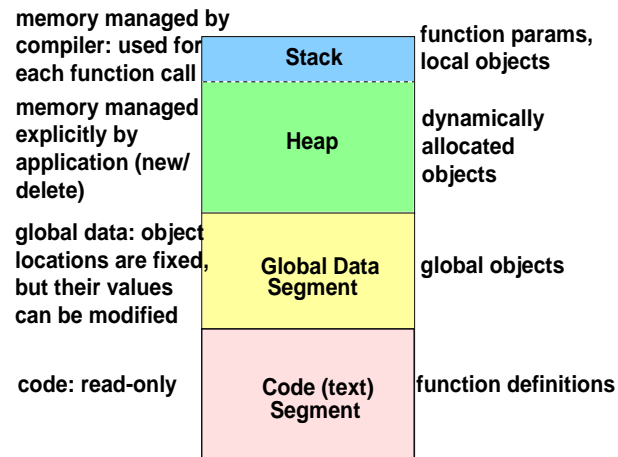
```
List_Node::List_Node (int i, List_Node *next)
: item_ (i),
  next_ (next)
{
}
```

Constructor contains  
 executable code

## Code and Data (cont'd)

- Header files contain (only!) *declarations*
- Implementation (.cc) files contain code and data *definitions*
- Code
  - Stored in (read-only) memory
  - All code stored in memory code (text) segment
- Data
  - Stored in writable (or read-only) memory
  - Location in memory depends on *allocation*

## Memory Allocation



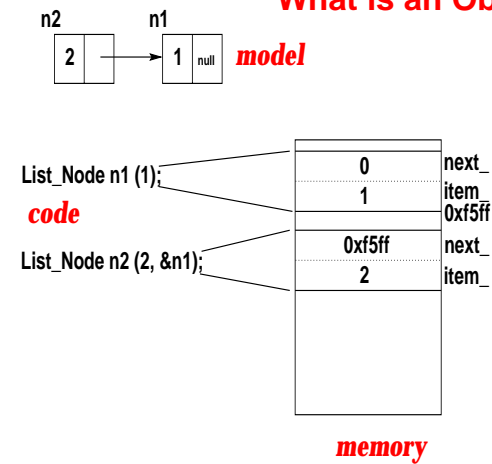
## A Note on Memory Access Errors

- *Segmentation violation*
  - Illegal access of a memory segment, *e.g.*,
    - \* Attempt to access memory outside of your text (code), data, and stack segments, such as at address 0
    - \* Attempt to write in the text (code) segment
- *Bus error*
  - Memory addresses apply to *words*, *e.g.*, 4 or 8 byte portions.
  - On many CPUs, integers, *etc.*, must be word-aligned for efficient memory access. Memory locations are accessed by word, not byte. So, you don't want to split integers, *etc.*, across words.
  - An attempt to access an integer, *etc.*, on a non-word boundary raises a bus error.

## A Note on Memory Access Errors, (cont'd)

- Bad pointer management can cause either error. So can many other code problems.
- To help track down the problem:
  - Try `make clean` (or `make realclean`) and a rebuild
  - Run the program in a debugger. It will stop when the problem is detected. If you're lucky that will be close to the code fault.
  - Note: be sure to delete individual objects with `delete`, and arrays with `delete []`. Mismatch can cause a bus error on the next *allocation*, because the allocator's free list may be corrupt.

## What is an Object?

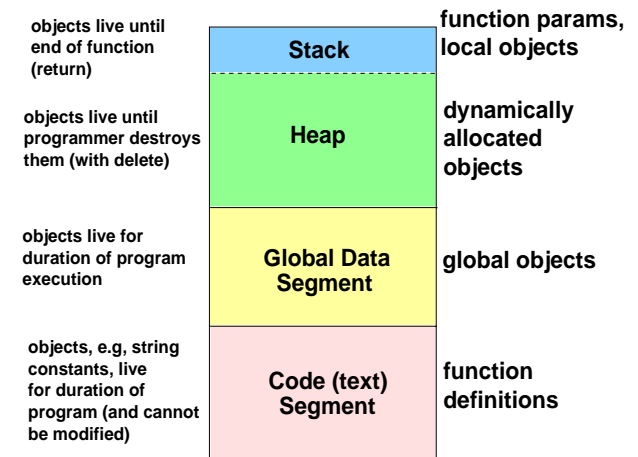


- In design: an entity in your *model* of the system.
- In source code: a (typed) variable.
- In object (compiled) code: an allocation of memory.
- In memory: a (named) portion of memory.

## How do I create an object?

- An object is created by creating an instance of its type.
  - That's why we call object creation *instantiation*
- In source code:
  - *Define* an object by creating a variable of its type, or
  - *define* an object by dynamically allocating it.
- In memory:
  - *Allocate* a portion of memory for the object.
  - If the object's class has a constructor, call it.

## Object Lifetime



## Dynamic Allocation

- Dynamically allocate an object that must be created within a function, but live past the end of the function call.

```
string *
string_factory (const char *contents) {
    string *s = new string (contents);
    return s;
}
```

- Another function should delete the dynamically allocated object when it is no longer needed. void func (string \*s) // [...] // Determined that s is no longer needed. delete s;
  - If it is never deleted, it is a *leak*.

## Where do I instantiate objects?

- Constants, such as string literals, are placed in the code by the compiler.
- Function parameters (passed by value) are placed on the stack.
- Local objects are placed on the stack.
- If an object outlives a function call, it must be allocated on the heap.
- Global objects are placed in global data.

## Instantiation Examples

- Local object

```
void func () {
    string s1 ("foo");
    // [...]
}
```

- Global object

```
- declaration
static history foo_history;
- definition
history Foo::foo_history ("Foo");
```

## Instantiation Examples (cont'd)

- Dynamically allocate an object on heap

```
string *s1 = new string ("foo");
```

- Refer to the objects' *address* via the pointer

```
string *a_string_address = s1;
```

- Refer to the *object* by *dereferencing* the pointer

```
string another_string = *s1;
```

- Destroy (deallocate) the object

```
delete s1;
```

## Objects, References, Pointers

- An object can have a name (*variable*)

```
string s1 ("foo");
```

- A unique object can have more than one name, but multiple *references* that can access it.

```
string &s2 = s1; // s2 and s1 refer to the same object!
```

- A reference cannot change the object that it refers to.
- A pointer contains the *address* of an object.

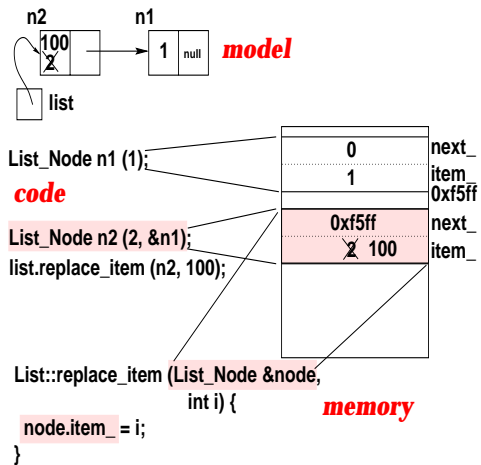
```
string *s3 = &s1; // *s3 refers to s1!
string s4 ("342");
s3 = &s4; // *s3 now refers to s4!
```

## References

- A reference refers to the same object as (*aliases*) another variable.
- A reference refers to an *object*, a pointer refers to an *address*.
- You can dereference a pointer to get a reference, but you must ensure that the pointer is valid.

```
int
func (Foo *fp)
{
    if (fp != 0) {
        Foo &f_ref = *fp;
        f_ref.update (); // same as fp->update ()
    }
}
```

## References (cont'd)



- References are often used for function parameters and return values.
- A parameter passed by reference refers to the object itself in the function call.

## Arrays

- A C++ array contains one or more objects of the same type.
 

```
char a[5] = {'a', 'b', 'c', 'd', '\0'};
```
- The size of an array object must be determined at compile time.
 

```
// The array size (5) was redundant.
char a[] = {'a', 'b', 'c', 'd', '\0'};
```
- The [] and \* notation for referring to arrays are interchangeable.
 

```
func (char argv[]); // Either the [] notation
func (char *argv); // or * notation is acceptable.
```

## Arrays (cont'd)

- The type of an array is the same as a pointer to the array's type.
 

```
char *b = a; // b aliases a,
char &c = a[0]; // and so does c (though
// this uncommon).
```
- The subscript operator [ ] dereferences the array pointer.
 

```
char e = a[1]; // e contains the value of
// a's second element.
```
- A C/C++ character string is a null-terminated array of chars.
 

```
char a[] = "abcd"; // Initialize with a string literal.
```

## Header Files

- Header files should only contain *declarations*
  - Class declarations
 

```
class List_Node {
public:
// Class interface (methods/functions)
List_Node (int,
List_Node * = 0); // Constructor
~List_Node (); // Destructor
// [...]
private:
// Class data members
int item_;
List_Node *next_;
};
```

## Header Files (Declarations, cont'd)

- In addition to class declarations, can contain:

- Function declarations

```
void toUpper(string & word);
void toLower(string & word);
```

- Data declarations

```
extern "C" {
    extern int errno;
}
```

- Incomplete (*forward*) class declarations.

```
class substring;
```

## Static

- Static has two meanings

- *Scope*: static file scope is only visible with the file. Avoid declaring objects outside of a class (static or not)!

```
// Foo.cc
static int i; // i can only be seen in Foo.cc.
```

- *Storage class*: static storage class is global.

```
class Foo {
    // [...]
    static history foo_history; // Global: all Foo
                                // instances share
                                // _one_ foo_history.
};
```

## Static (cont'd)

- To define a static data member:

```
// Foo.cc
history Foo::foo_history ("Foo");
```

- To declare a static method:

```
// Foo.h
class Foo {
    // [...]
    static Foo &instance ();
```

## Static (cont'd)

- To define a static method:

```
// Foo.cc
Foo &Foo::instance ()
{
    Foo *instance = new Foo ();
    return *instance;
}
```

## Things to watch out for

- Be careful with aliases (references) and pointers!
- Never use an object after it has been deleted from the heap!

```
string *s = new string;
// [...]
delete s;
```

```
string t = toUpper (*s); // BAD: s was deleted!
```

- Never `delete` an object that you didn't allocate with `new`!

## Things to watch out for (cont'd)

- Never pass the address of a local object *out* of a function!

```
string &
func () {
    string local_string ("foo");

    [...]

    return local_string; // BAD: never return reference
                        // to local object!
}
```

## Things to watch out for (cont'd)

- Avoid dynamic allocation of local variables, unless lifetime must extend past function end.

```
string *s1 = new string ("foo");
cout << *s1 << end;
delete s1;
```

is identical to:

```
string s1 ("foo");
cout << s1 << end;
```

## Things to watch out for (cont'd)

- Minimize contents of header files.

```
#ifndef SUBSTRING_H
#define SUBSTRING_H

class string; // forward declaration,
              // instead of #include

class substring
{
public:
    substring (string & base, int start, int length);
    // [...]
}
```