COMP322 - Introduction to C++

Winter 2011

Lecture 4 - Memory Management

Milena Scaccia

School of Computer Science
McGill University

January 25, 2011

Memory management in C++

Provides ways to allocate portions of memory to programs at their request, and freeing it for reuse when it is no longer needed.

When your program runs, it has access to certain portions of memory:

- Stack: Section of computer memory where all variables declared and initialized before runtime are stored.
- Heap (aka free store): Section of computer memory where all the variables created or initialized at runtime are stored. The pattern of allocation and size of blocks is not known until run time (Dynamic memory allocation).

Memory Layout

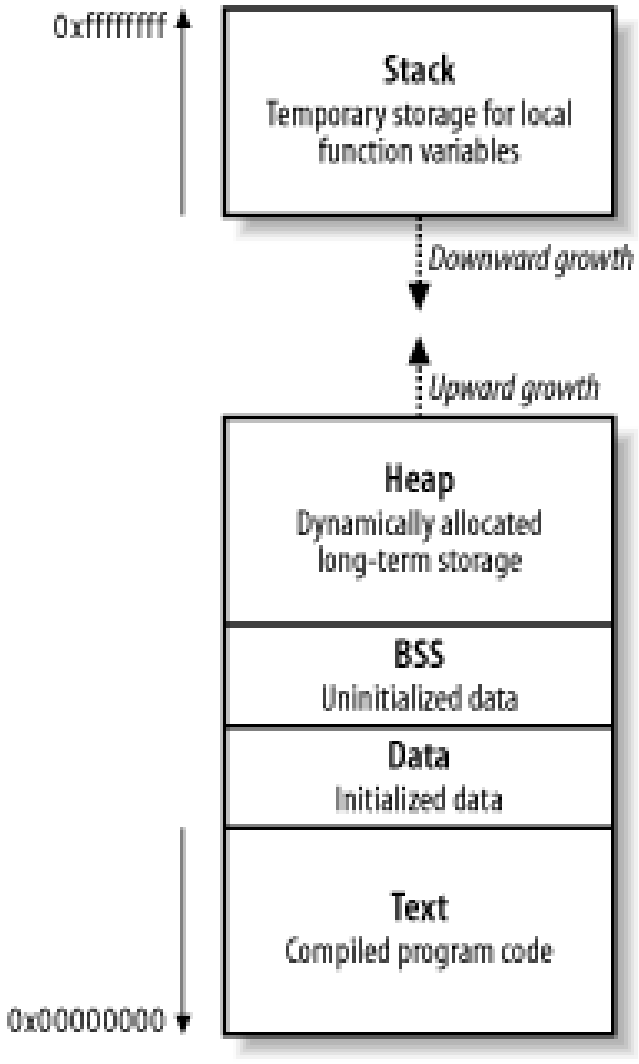


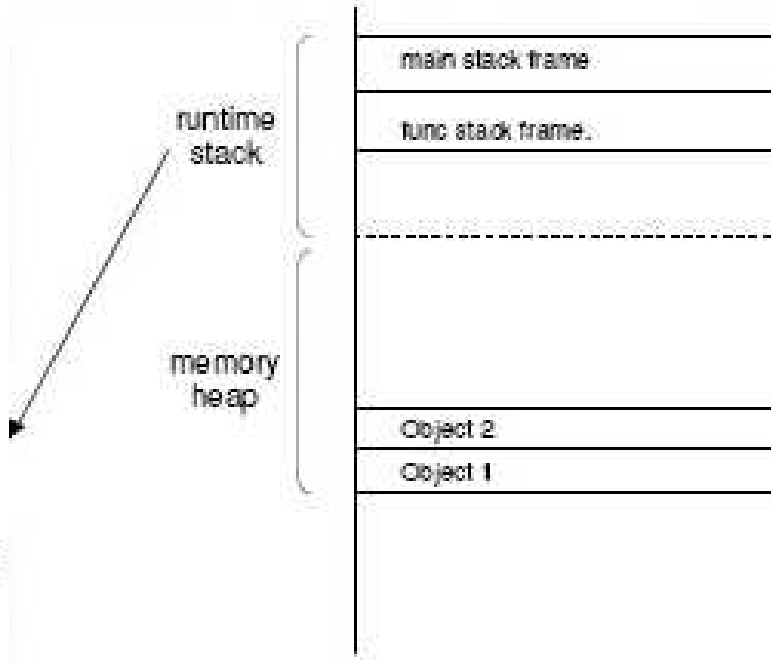
Figure: Computer Memory Source: `etutorials.org/Networking/`

Computer Memory Example: Stack Allocation

All variables created so far are allocated on the runtime stack

```
double sqrt_by_newton(double x) {
  double delta;
  double y = x;                 // initial guess

  do {
    double new_y = y - (y * y - x) / (2 * y);
    delta = y - new_y;
    y = new_y;
  } while (delta > 1e-12);   // new_y deleted here
  return (y);
} // delta and y deleted here
```

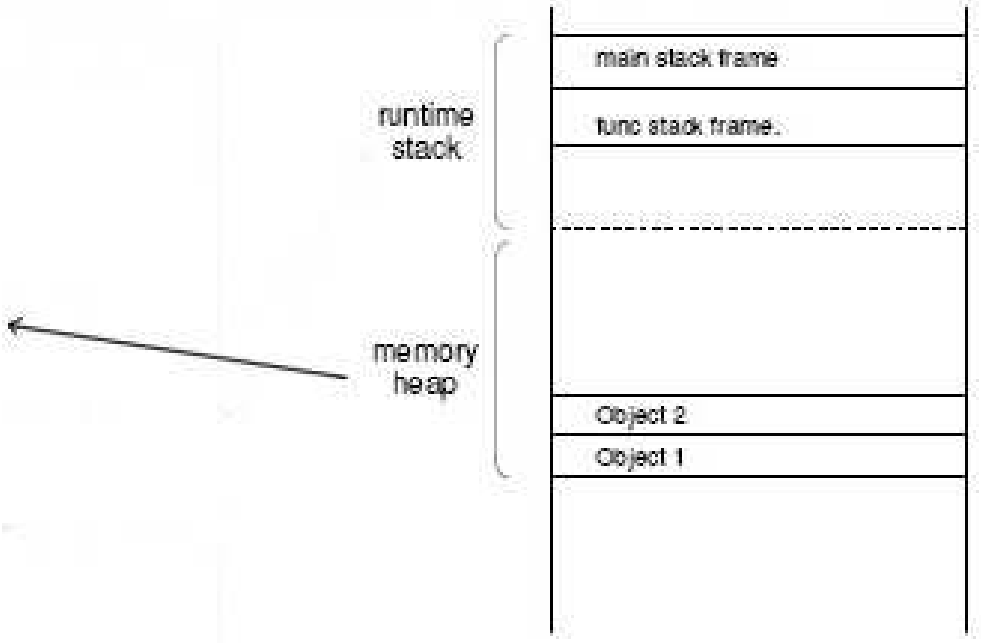


All allocated data is reclaimed when the functon exits.

Computer Memory Example: Heap Allocation

```
int main(){
  int *array = new int[5];
  // ...
  delete[] array;

  return 0;
}
```



With new and delete, we have to manage memory ourselves.

Importance of Memory Management

Correctly managing memory is crucial. Failing to do so may result in serious consequences.

Your program may:

- Crash (segfaults)
- Behave unexpectedly
- Run slowly (memory hog)
- Be vulnerable to hackers

In this lecture:

- How to allocate/deallocate memory
- Common memory management errors and how to prevent them

The free store

C:

- `malloc()`
- `free()`

C++

- `new`
- `delete`

The free store in C

The free store, or heap, allows us to create objects whose addresses may be assigned to pointer variables.

The C heap is accessed by the functions `malloc()` and `free()`.

- `void *malloc(size_t nbytes)` returns a pointer to a region of memory at least 'nbytes' in size. It returns NULL if no such region is available.
- `void free(void *ptr)` marks the region as unused.

An pointer returned by `malloc()` can be used until it is explicitly released by calling `free()`.

The free store in C

- The argument to `malloc()` is the number of bytes required, often a `sizeof` expression.
- The memory returned is *not* initialized in any way.
- `malloc()` returns `void *`, so we cast the return value to the required pointer type.
- Any argument to `free()` must have been returned by `malloc()`

```
struct student* create_student(long id, double grade) {
    struct student *new_p ;
    new_p = (struct student*) malloc(sizeof(struct student));
    if (new_p != NULL) {
      new_p->sID = id;
      new_p->sGrade = grade;
   }
   return (new_p);
}

void delete_student(struct student *sp) {
   free(sp); // Memory is no longer in use
}
```

The free store in C++

While `malloc()` and `free()` remain part of the C++ standard library, C++ introduces two operators which manipulate the free store.

- `new` - Allocate memory on the free store.
- `delete` - Free memory if the argument is non-zero.

```
struct student *create_student(long id, double grade) {
  struct student *new_p = new struct student;
  new_p->sID = id;
  new_p->sGrade = grade;
  return (new_p);
}

void delete_student(struct student *sp) {
  delete sp; // Memory is no longer in use
}
```

Details of the `new` operator

- A `new` expression implicitly calls the *constructor* for an object, which may initialize the object.
- However, the initial value of the memory is undefined.
- We can specify initial arguments for the constructor:

```
double *pd1 = new double(1.1); // Sets the initial value
```

- If a `new` operation fails, it throws an *exception*. By default this will end the program.
- We can suppress the exception with the `nothrow` parameter:

```
#include <new>
symbol *sym_p = new(std::nothrow) symbol();
if (sym_p == NULL) { // Allocation failed
```

Creating and deleting arrays

The `new` and `delete` operators also work with arrays. This allows us to set the lengths of arrays at runtime. We can access elements using the [] operator, indexing in an array.

```
int *create_array(int size) {
   int *array = new int[size]; // Allocate 'size' integers
   // perform some initialization
   for(int i = 0; i<size; i++)
      array[i] = 0;
   return array;
}

void delete_array(int *array) {
  delete [] array; // brackets tell delete this is an array
}
```

We must specify `delete []` when deleting an array.

Creating multidimensional arrays

The syntax of C++ does not allow for easy creation of multidimensional arrays:

```
float **matrix = new float[10][10]; // Illegal!
```

Instead we have to use a more complex initialization:

```
float **matrix = new float *[10];
int i, j;
for (i = 0; i < 10; i++) {
  matrix[i] = new float[10];
}
// now we can access matrix[i][j]
```

Many of these sort of things are better handling using the standard library.

Advantages of C++ memory management

The `new` and `delete` operators provide several advantages over `malloc()` and `free()`.

1. No need to cast - New automatically returns a pointer of the correct type.
2. No need to explicitly calculate the size of the object.
3. Initialization is performed if a constructor is defined.
4. Can throw an exception on failure, which can simplify error handling.
5. Will call the *destructor* before deleting an object, if defined.
6. You can override the `new` operator and provide your own implementation for debugging or other special purposes.

Common memory management error 1

There are a large number of ways in which memory management can go wrong.

What is wrong with the following code fragment?

```
int n;
int *p = &n;
// ...
delete p;

int *pv = new int[5];
pv++;
delete [] pv;
```

Deleting a pointer that was not returned by `new`:

```
int n;
int *p = &n;
// ...
delete p;           // error!

int *pv = new int[5];
pv++;               // alters the pointer
delete [] pv;       // error!
```

Common memory management error 2

What is wrong with the following code fragment?

```
float *pv1 = new float[size];
float sum;

sum += pv1[0];
```

Assuming the memory is initialized

```
float *pv1 = new float[size];
float sum;

sum += pv1[0];   // The value of pv1[0] is undefined!
```

Common error 3: Memory Leak

mysticmemos.wordpress.com

Memory Leak: Failing to delete allocated memory. This is a common error in programming with languages that have no built-in garbage collection.

```
char *linebuf = new char[1024];

// ...

linebuf = new char[128]; // Another object

// ...

delete [] linebuf;      // deleted 2nd, but not 1st object
```

Memory Leak cont'd

No serious consequences for programs that run over a short period of time.

Examples of when memory leaks can be serious:

- The program runs for an extended time and consumes additional memory over time (e.g. background tasks on servers)
- Memory is very limited (e.g. in embedded systems)

Results in: Diminished computer performance/system slow downs/application failure

Common error 4: Buffer overflow

Buffer overflow: trying to put more data into an array than there is room for. As a result, the program may overrun the buffer's boundary and overwrite adjacent memory.

- C/C++ provide no built-in protection against accessing or overwriting data in any part of memory
- No automatic bounds checking for arrays $\Longrightarrow$ this is the responsibility of the programmer!

Security issue! "Giving a program more data than it can handle is the number one trick in the arsenal of the hacker." [1]

[1] Fred Swartz, MIT License

Recent exploitations of buffer overflow

- Internet worms compromised a large number of systems by exploiting buffer overflows:
  - Code Red worm
  - SQL Slammer worm
- Buffer overflows in licensed games have been exploited to allow unlicensed software to run on the console without the need for hardware modifications [2].

[2] gamesindustry.biz

Common memory management error 5

What is wrong with the following code fragment?

```
int *p = new int(10);
// ...
delete p;
// ...
delete p;
```

Multiple erroneous memory de-allocations

```
int *p = new int(10); // Initialize *p==10
// ...
delete p;
// ...
delete p;          // error!
```

- Programs may erroneously free memory that has already been freed
- This can result in an immediate crash or be exploitable, allowing an attacker to cause arbitrary code to be executed [3].

[3] David A. Wheeler, Secure Programming for Linux and Unix HOWTO, March 2003