C Review Session

Hosted by: Collin Johnston and Maajid Nazrulla

http://bit.ly/MKzj0f

Introduction: Welcome!

According to Wikipedia, C is a general purpose, statically typed, imperative (procedural), multiplatform language initially developed by Dennis Ritchie during the late 1960s and early 1970s at AT&T Bell Labs.

This review session will cover basics of ANSI C, a family of standards published by the American National Standards Institute.

For the purposes of this session it is expected that you have basic knowledge from 61B and 61C of a statically typed language such as Java, and that you have basic programming experience already.

Data Types and Sizes

Type	Explanation
`char`	smallest addressable unit of the machine that can contain basic character set. It is an integer type. Actual type can be either signed or unsigned depending on the implementation.
`signed char`	same size as `char`, but guaranteed to be signed.
`unsigned char`	same size as `char`, but guaranteed to be unsigned.

`short`
`short int`
`signed short`
`signed short int`
	short signed integer type. At least 16 bits in size.

`unsigned short` `unsigned short int`	same as `short`, but unsigned.
`int` `signed int`	basic signed integer type. At least 16 bits in size.
`unsigned` `unsigned int`	same as `int`, but unsigned.

`long`
`long int`
`signed long`
`signed long int`
	long signed integer type. At least 32 bits in size.

`unsigned long` `unsigned long int`	same as `long`, but unsigned.

`long long`
`long long int`
`signed long long`
`signed long long int`
	long long signed integer type. At least 64 bits in size. Specified since the C99 version of the standard.

`unsigned long long` `unsigned long long int`	same as `long long`, but unsigned. Specified since the C99 version of the standard.
`float`	single precision floating-point type. Actual properties unspecified (except minimum limits), however on most systems this is the IEEE 754 single-precision binary floating-point format. This format is required by the optional Annex F "IEC 60559 floating-point arithmetic".
`double`	double precision floating-point type. Actual properties unspecified (except minimum limits), however on most systems this is the IEEE 754 double-precision binary floating-point format. This format is required by the optional Annex F "IEC 60559 floating-point arithmetic".
`long double`	extended precision floating-point type. Actual properties unspecified. Unlike types `float` and `double`, it can be either 80-bit floating point format, the non-IEEE "double-double" or IEEE 754 quadruple-precision floating-point format if a higher precision format is provided, otherwise it is the same as `double`. See the article on long double for details.

Table taken from Wikipedia.

Operator Precedence in C

Operator Precedence Chart

Operator Type	Operator	Associativity
Primary Expression Operators	() [] . -> *expr*++ *expr*--	left-to-right
Unary Operators	* & + - ! ~ ++*expr* --*expr* (*typecast*) sizeof	right-to-left
Binary Operators	* / %	left-to-right
	+ -	
	>> <<	
	< > <= >=	
	== !=	
	&	
	^	
	|	
	&&	
	||	
Ternary Operator	?:	right-to-left
Assignment Operators	= += -= *= /= %= >>= <<= &= ^= |=	right-to-left
Comma	,	left-to-right

Note that operator overloading is not supported in C, beyond what is natively implemented.

Table taken from http://www.swansontec.com/sopc.html .

Operator Precedence in C

Operator Precedence Chart

<table>
<tr><th>Operator Type</th><th>Operator</th><th>Associativity</th></tr>
<tr><td>Primary Expression Operators</td><td><code>() [] . -></code> expr++ expr--</td><td>left-to-right</td></tr>
<tr><td>Unary Operators</td><td><code>* & + - ! ~</code> ++expr --expr (typecast) <code>sizeof</code></td><td>right-to-left</td></tr>
<tr><td rowspan="10">Binary Operators</td><td><code>* / %</code></td><td rowspan="10">left-to-right</td></tr>
<tr><td><code>+ -</code></td></tr>
<tr><td><code>>> <<</code></td></tr>
<tr><td><code>< > <= >=</code></td></tr>
<tr><td><code>== !=</code></td></tr>
<tr><td><code>&</code></td></tr>
<tr><td><code>^</code></td></tr>
<tr><td><code>|</code></td></tr>
<tr><td><code>&&</code></td></tr>
<tr><td><code>||</code></td></tr>
<tr><td>Ternary Operator</td><td><code>?:</code></td><td>right-to-left</td></tr>
<tr><td>Assignment Operators</td><td><code>= += -= *= /= %= >>= <<= &= ^= |=</code></td><td>right-to-left</td></tr>
<tr><td>Comma</td><td><code>,</code></td><td>left-to-right</td></tr>
</table>

Note that operator overloading is not supported in C, beyond what is natively implemented.

Suppose we have two unsigned ints, `lo` and `hi`, between 0 and 255 and we want to set a third unsigned integer to a 16 bit value whose lower order bits are `lo` and whose higher order bits are those of `hi`.

Table taken from http://www.swansontec.com/sopc.html .

Operator Precedence in C

Operator Precedence Chart

<table>
<tr><th>Operator Type</th><th>Operator</th><th>Associativity</th></tr>
<tr><td>Primary Expression Operators</td><td>() [] . -> expr++ expr--</td><td>left-to-right</td></tr>
<tr><td>Unary Operators</td><td>* & + - ! ~ ++expr --expr (typecast) sizeof</td><td>right-to-left</td></tr>
<tr><td rowspan="11">Binary Operators</td><td>* / %</td><td rowspan="11">left-to-right</td></tr>
<tr><td>+ -</td></tr>
<tr><td>>> <<</td></tr>
<tr><td>< > <= >=</td></tr>
<tr><td>== !=</td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>Ternary Operator</td><td>?:</td><td>right-to-left</td></tr>
<tr><td>Assignment Operators</td><td>= += -= *= /= %= >>= <<= &= ^= |=</td><td>right-to-left</td></tr>
<tr><td>Comma</td><td>,</td><td>left-to-right</td></tr>
</table>

Table taken from http://www.swansontec.com/sopc.html .

Note that operator overloading is not supported in C, beyond what is natively implemented.

Suppose we have two unsigned ints, `lo` and `hi`, between 0 and 255 and we want to set a third unsigned integer to a 16 bit value whose lower order bits are `lo` and whose higher order bits are those of `hi`.

We choose to do:

```
unsigned int16_t i = hi << 8 + lo;
```

What is wrong with this?

Operator Precedence in C

Operator Precedence Chart

Operator Type	Operator	Associativity
Primary Expression Operators	`() [] . -> expr++ expr--`	left-to-right
Unary Operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	right-to-left
Binary Operators	`* / %`	left-to-right
	`+ -`	
	`>> <<`	
	`< > <= >=`	
	`== !=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary Operator	`?:`	right-to-left
Assignment Operators	`= += -= *= /= %= >>= <<= &= ^= |=`	right-to-left
Comma	`,`	left-to-right

Table taken from http://www.swansontec.com/sopc.html .

Note that operator overloading is not supported in C, beyond what is natively implemented.

Suppose we have two unsigned ints, `lo` and `hi`, between 0 and 255 and we want to set a third unsigned integer to a 16 bit value whose lower order bits are `lo` and whose higher order bits are those of `hi`.

We choose to do:

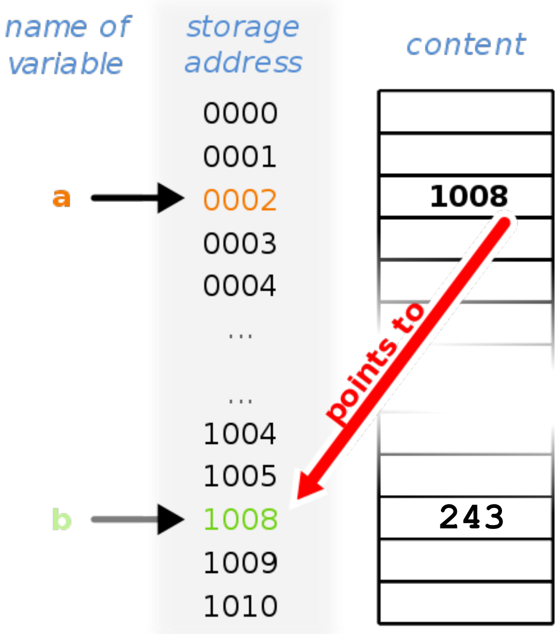
```
unsigned int16_t i = hi << 8 + lo;
```

Instead, we choose to do the following:

```
unsigned int16_t i = hi << 8 | lo;
```

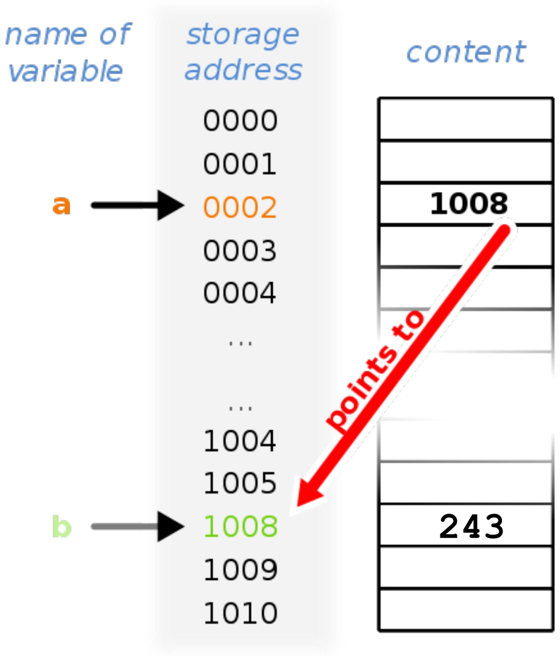
Is there anything wrong now?

Pointers



- Consider memory to be a single huge array
  - Each cell/entry of the array has an address
  - Each cell also stores some value
- Don’t confuse the address referring to a memory location with the value stored there

Pointers



- Syntax:

```
a = 1008
*a = 243
&b = 1008
&a = ?

a[4] = *(a+4)
```

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  int* p;
  int a = 7;
  p = &a;
  int i = *p;
  printf("%u\n", &i);
  printf("%d\n", i);
  return 0;
}
```

What does this code do?

What does this code do?

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
  int* p;       // Declares a pointer to an int
  int a = 7;
  p = &a;
  int i = *p;
  printf("%u\n", &i);
  printf("%d\n", i);
  return 0;
}
```

What does this code do?

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  int* p;       // Declares a pointer to an int
  int a = 7;    // Declares an int with value 7
  p = &a;
  int i = *p;
  printf("%u\n", &i);
  printf("%d\n", i);
  return 0;
}
```

What does this code do?

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  int* p;       // Declares a pointer to an int
  int a = 7;    // Declares an int with value 7
  p = &a;       // sets the value of p to the address of a
  int i = *p;
  printf("%u\n", &i);
  printf("%d\n", i);
  return 0;
}
```

What does this code do?

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
  int* p;         // Declares a pointer to an int
  int a = 7;      // Declares an int with value 7
  p = &a;         // sets the value of p to the address of a
  int i = *p;     // Declares an int i whose value is the value at address p
  printf("%u\n", &i);
  printf("%d\n", i);
  return 0;
}
```

What does this code do?

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  int* p;        // Declares a pointer to an int
  int a = 7;     // Declares an int with value 7
  p = &a;        // sets the value of p to the address of a
  int i = *p;    // Declares an int i whose value is the value at address p
  printf("%u\n", &i); // Prints the address of i
  printf("%d\n", i);
  return 0;
}
```

What does this code do?

```
#include <stdio.h>
int main(int argc, char* argv[]) {
  int* p;        // Declares a pointer to an int
  int a = 7;     // Declares an int with value 7
  p = &a;        // sets the value of p to the address of a
  int i = *p;    // Declares an int i whose value is the value at address p
  printf("%u\n", &i); // Prints the address of i
  printf("%d\n", i);  // Prints the value of i
  return 0;
}
```

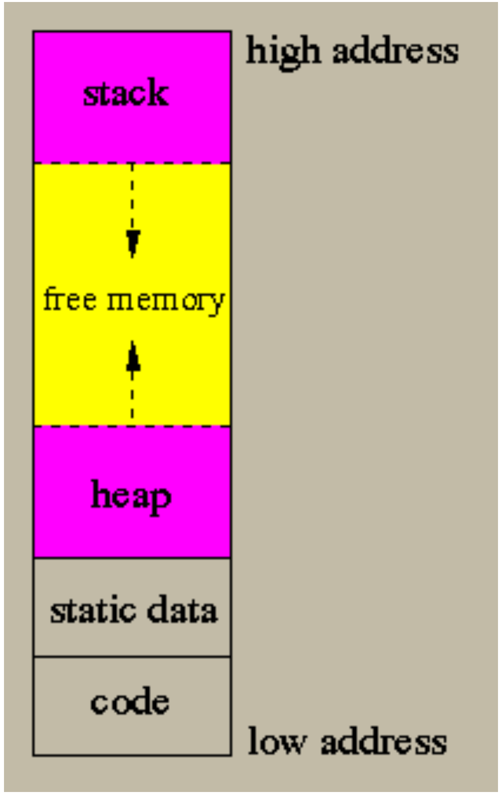
What does this code do?

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  int* p;        // Declares a pointer to an int
  int a = 7;     // Declares an int with value 7
  p = &a;        // sets the value of p to the address of a
  int i = *p;    // Declares an int i whose value is the value at address p
  printf("%u\n", &i); // Prints the address of i
  printf("%d\n", i);  // Prints the value of i (7)
  return 0;
}
```

Memory Basics

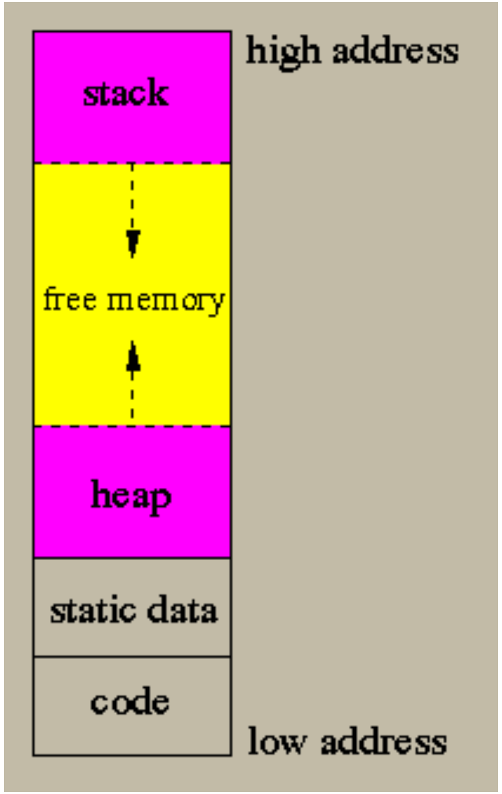
Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html



Memory Basics

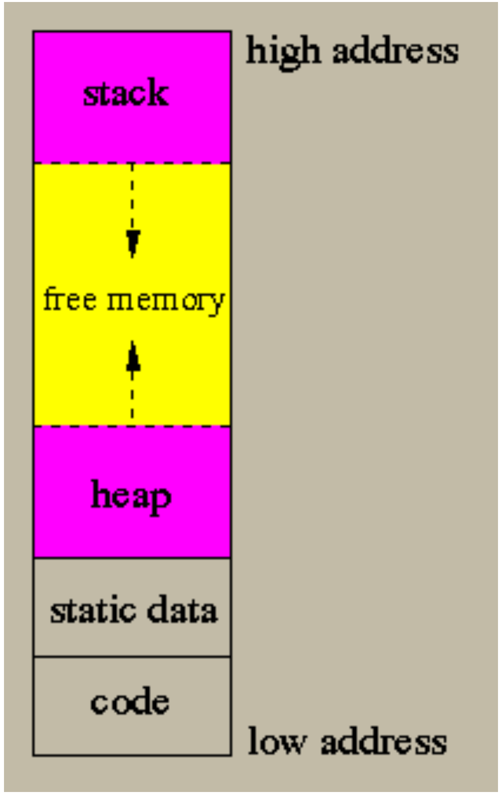
Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

We have four regions in the address space of a program. They are, from highest address to lowest:



Memory Basics

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

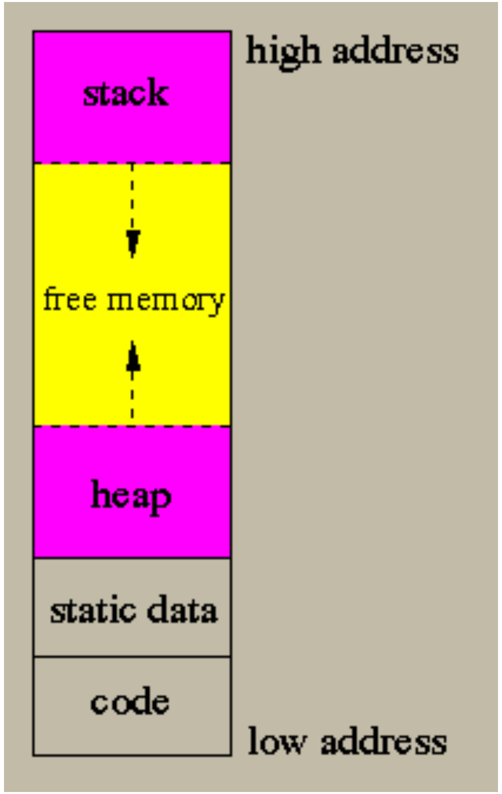


We have four regions in the address space of a program. They are, from highest address to lowest:

- `Stack` (grows downward, toward lower addresses)

Memory Basics

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html



We have four regions in the address space of a program. They are, from highest address to lowest:

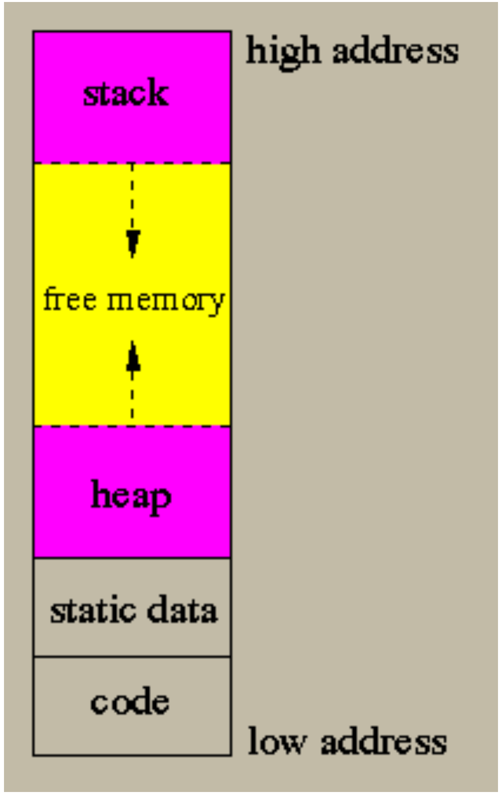
- `Stack` (grows downward, toward lower addresses)
- `Heap` (grows upward, resizes dynamically)

Memory Basics

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

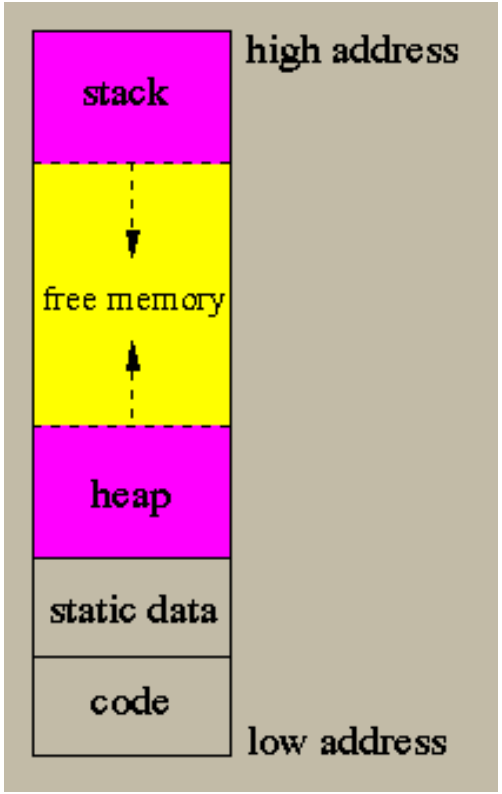
We have four regions in the address space of a program. They are, from highest address to lowest:

- `Stack` (grows downward, toward lower addresses)
- `Heap` (grows upward, resizes dynamically)
- `Static` (doesn't change in size)



Memory Basics

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html



We have four regions in the address space of a program. They are, from highest address to lowest:

- `Stack` (grows downward, toward lower addresses)
- `Heap` (grows upward, resizes dynamically)
- `Static` (doesn't change in size)
- `Code` (doesn't change)

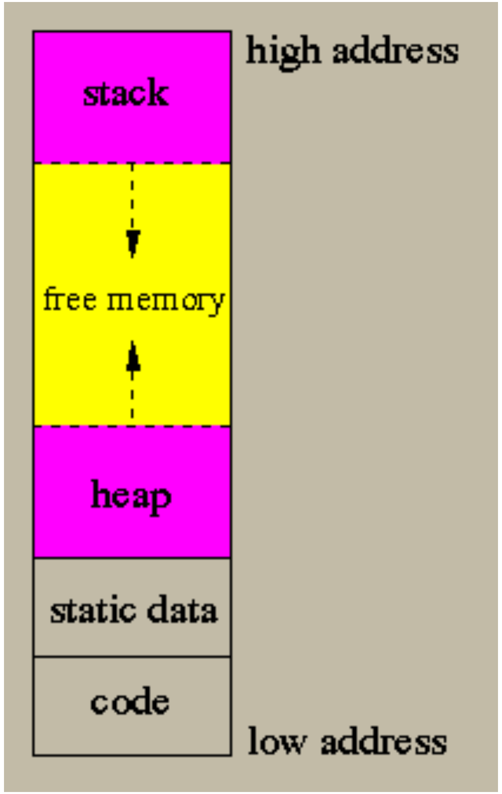
Memory Basics

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

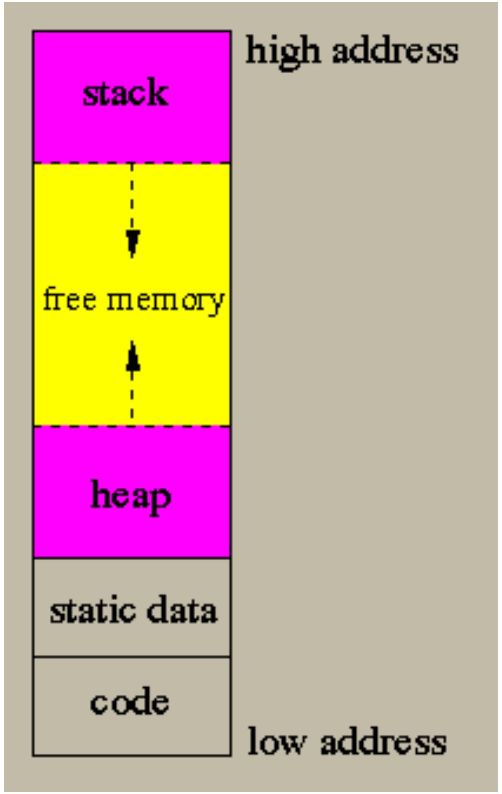
We have four regions in the address space of a program. They are, from highest address to lowest:

- `Stack` (grows downward, toward lower addresses)
- `Heap` (grows upward, resizes dynamically)
- `Static` (doesn't change in size)
- `Code` (doesn't change)

Let's cover each briefly, starting with the `code` section.

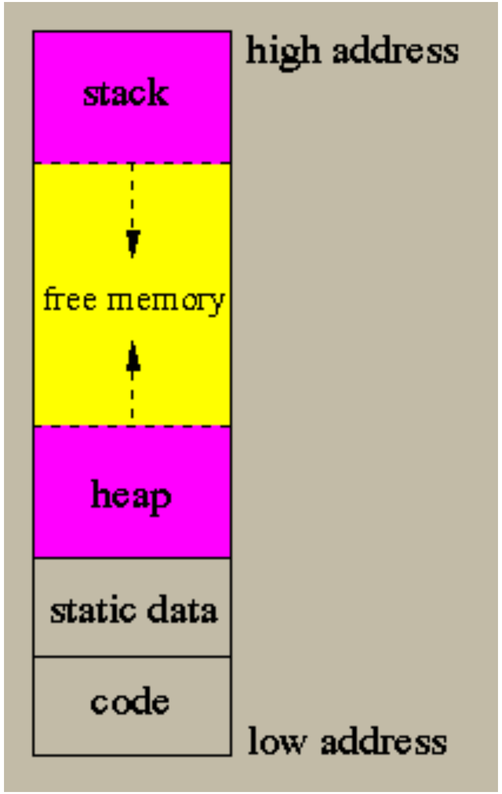


Memory Basics: Code Section



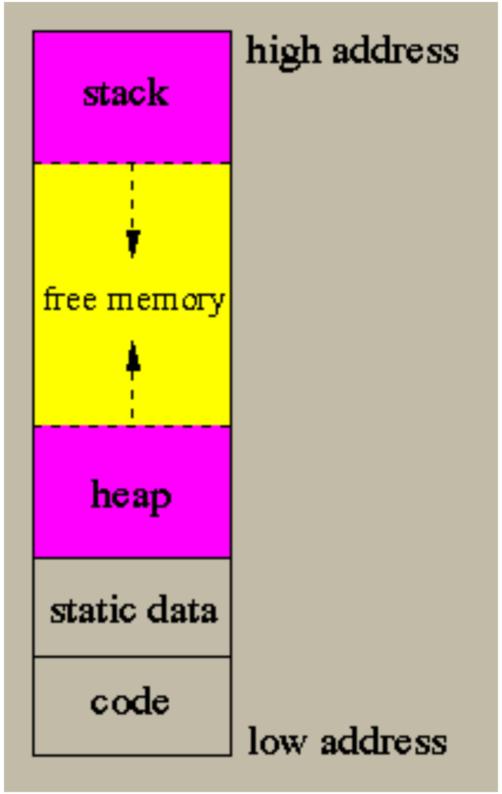
Memory Basics: Code Section

- The code section is where the C source code from your program resides in memory.



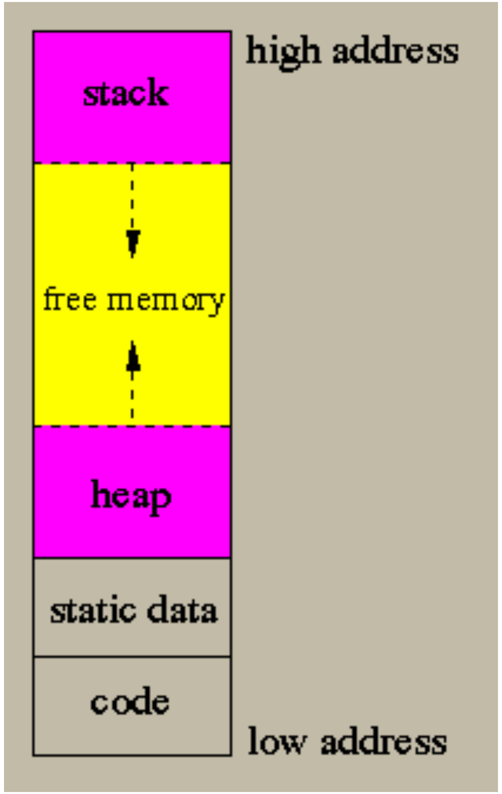
Memory Basics: Code Section

- The code section is where the C source code from your program resides in memory.
- It is also alternatively known as the code segment, text segment, or just text.



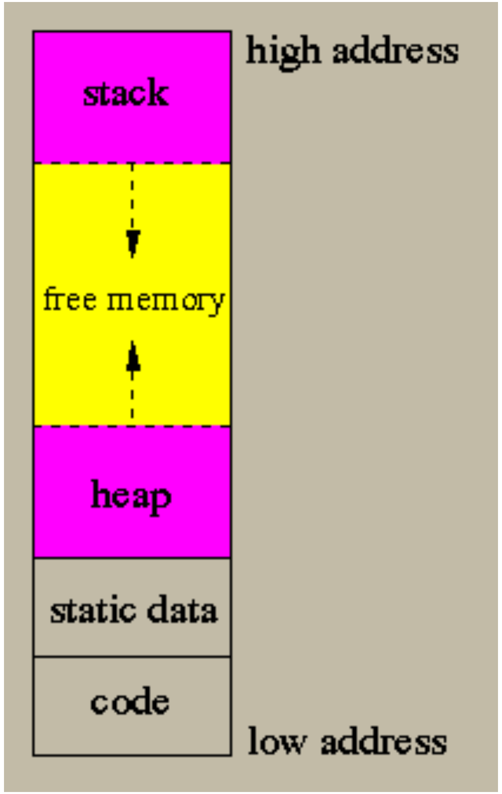
Memory Basics: Code Section

- The code section is where the C source code from your program resides in memory.
- It is also alternatively known as the code segment, text segment, or just text.
- This section is allocated at run time and is a fixed size.



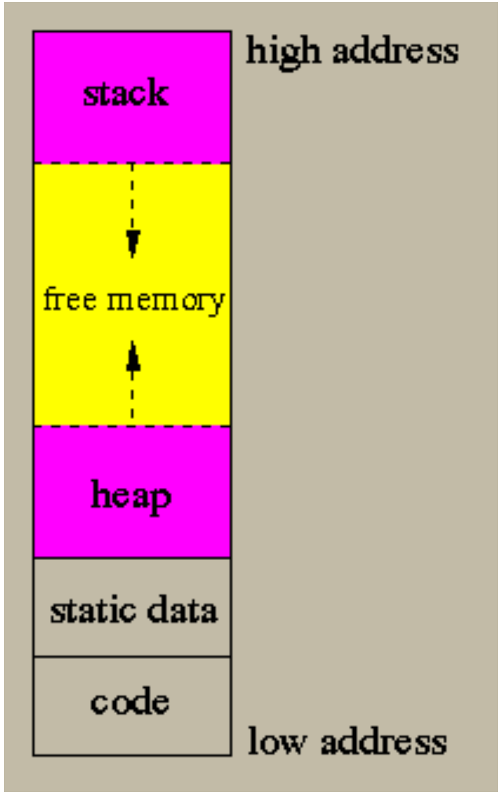
Memory Basics: Code Section

- The code section is where the C source code from your program resides in memory.
- It is also alternatively known as the code segment, text segment, or just text.
- This section is allocated at run time and is a fixed size.
- Generally the code segment is read-only. Architectures in which the code segment is not read-only support self-modifying code.

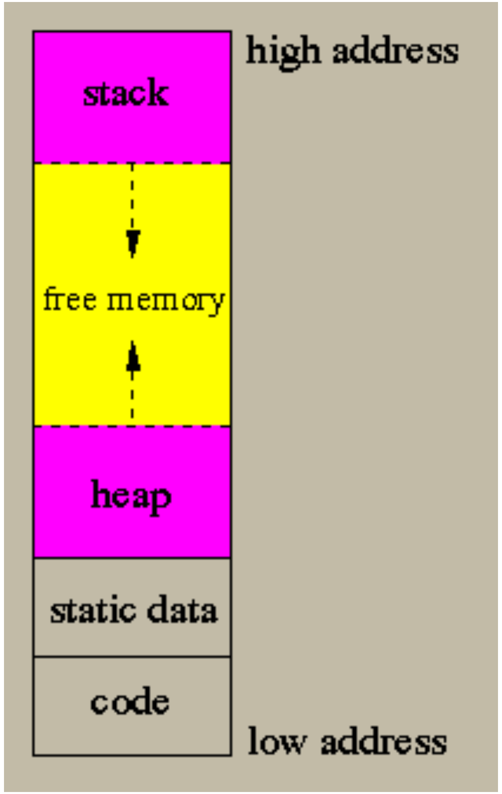


Memory Basics: Code Section

- The code section is where the C source code from your program resides in memory.
- It is also alternatively known as the code segment, text segment, or just text.
- This section is allocated at run time and is a fixed size.
- Generally the code segment is read-only. Architectures in which the code segment is not read-only support self-modifying code.
- The code section is often placed below the heap and stack locations to protect it from being overwritten due to heap or stack overflows.

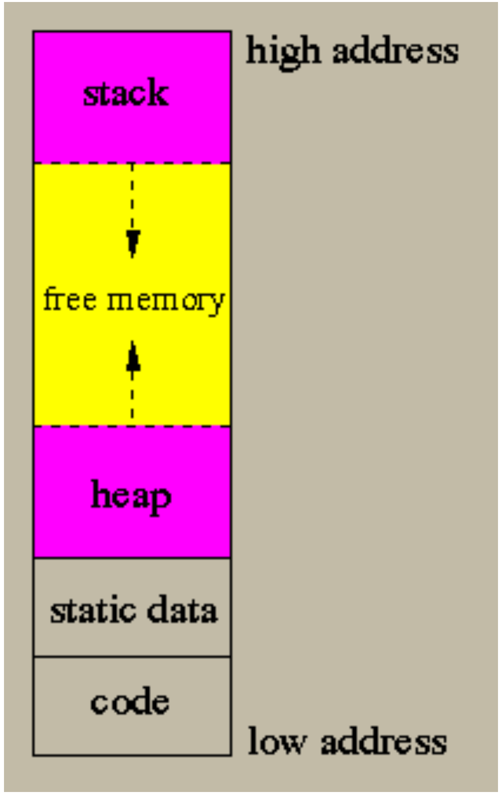


Memory Basics: Static Data Section



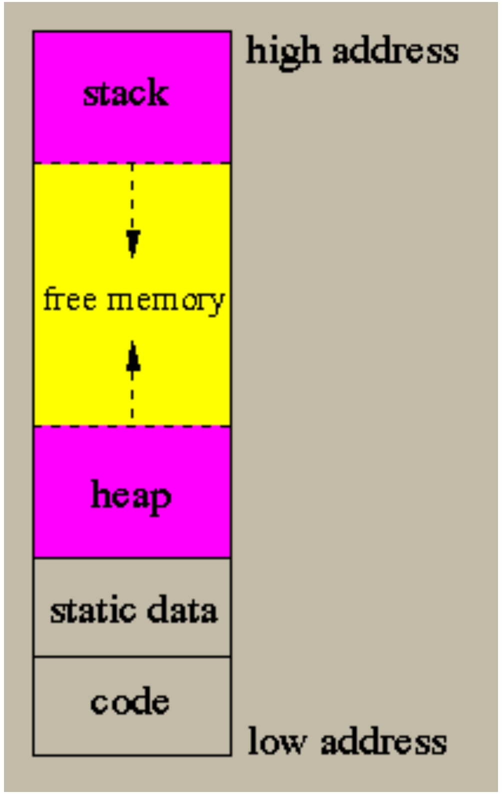
Memory Basics: Static Data Section

- The static data section is where persistent variables such as global variables (or any variables declared outside a function) and string literals are stored.



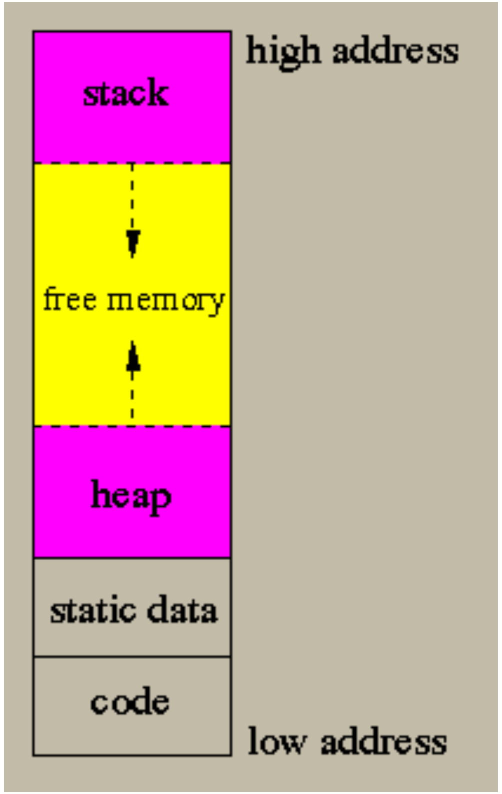
Memory Basics: Static Data Section

- The static data section is where persistent variables such as global variables (or any variables declared outside a function) and string literals are stored.
- The data in the static data section can change, but the size is determined at compile time and cannot change.



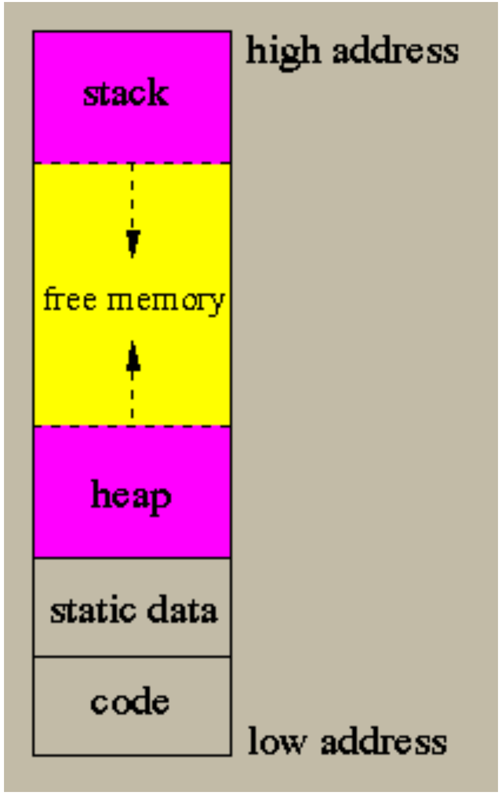
Memory Basics: Static Data Section

- The static data section is where persistent variables such as global variables (or any variables declared outside a function) and string literals are stored.
- The data in the static data section can change, but the size is determined at compile time and cannot change.
- Like the code section, static data is often placed below the heap and stack locations to protect it from being overwritten due to heap or stack overflows.



Memory Basics: Heap

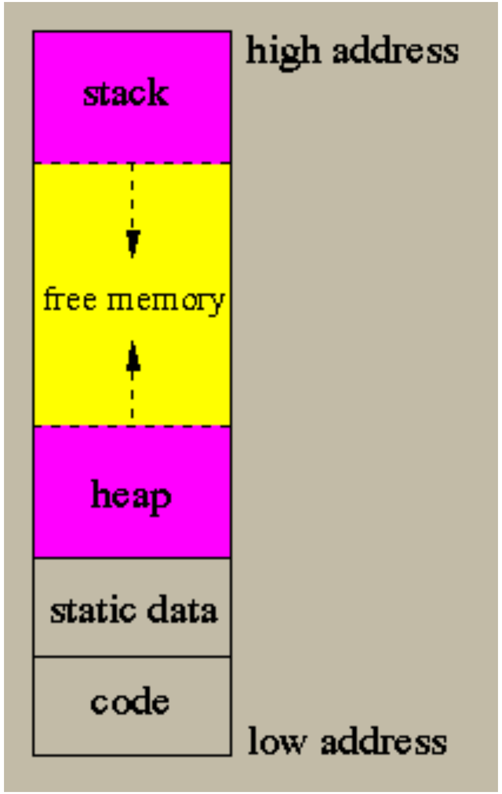
Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html



Memory Basics: Heap

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

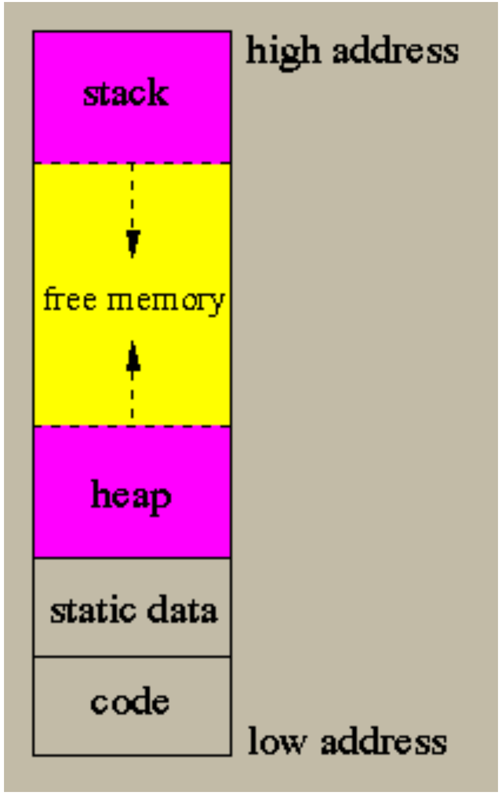
- The heap segment is where dynamically allocated data resides.



Memory Basics: Heap

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

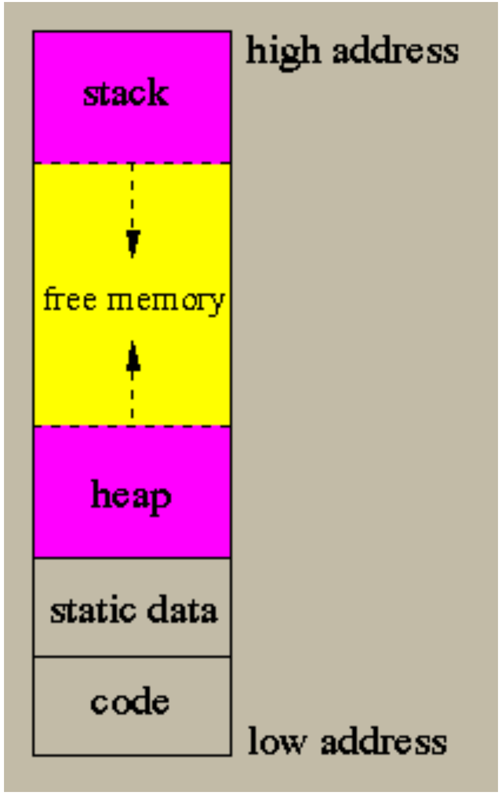
- The heap segment is where dynamically allocated data resides.
- Addressing for the heap segment generally starts above the static data section and grows upward. This is done to maximize the amount of memory available for dynamic allocation while minimizing interference with the stack.



Memory Basics: Heap

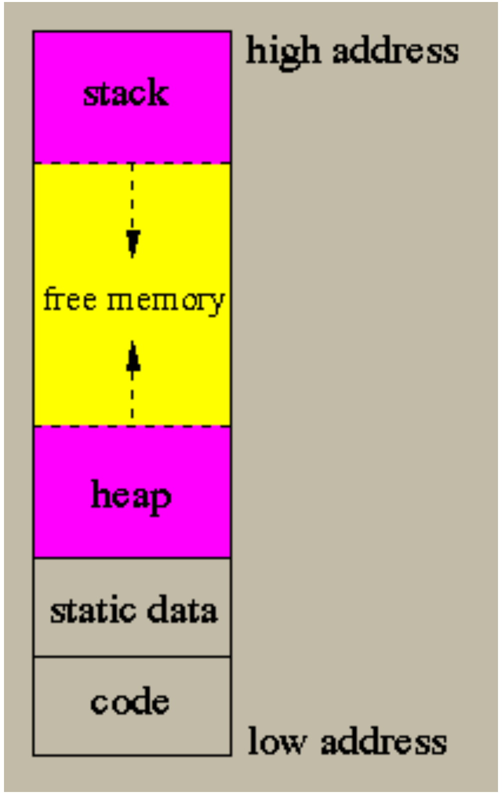
- The heap segment is where dynamically allocated data resides.
- Addressing for the heap segment generally starts above the static data section and grows upward. This is done to maximize the amount of memory available for dynamic allocation while minimizing interference with the stack.
- The programmer must manage the heap in C-this is done through several functions which we will soon cover.

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html



Memory Basics: Heap

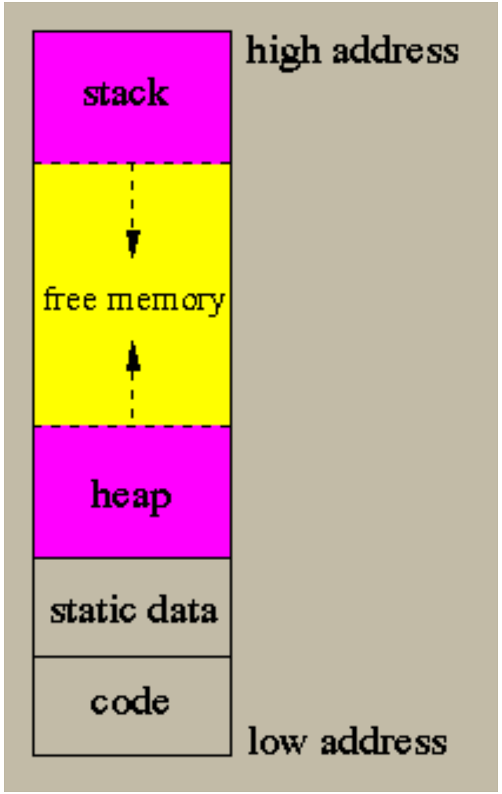
Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html



- The heap segment is where dynamically allocated data resides.
- Addressing for the heap segment generally starts above the static data section and grows upward. This is done to maximize the amount of memory available for dynamic allocation while minimizing interference with the stack.
- The programmer must manage the heap in C-this is done through several functions which we will soon cover.
- The data in the heap can be accessed across functions. This is useful for data structures that require the flexibility of dynamic memory allocation as well as access by multiple functions.

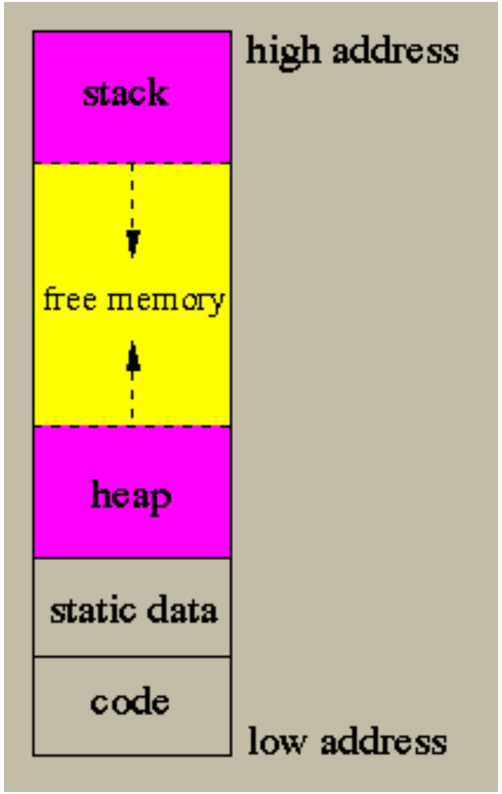
Memory Basics: Stack

Image taken from:
http://lambda.uta.edu/cse5317/notes/node33.html



Memory Basics: Stack

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

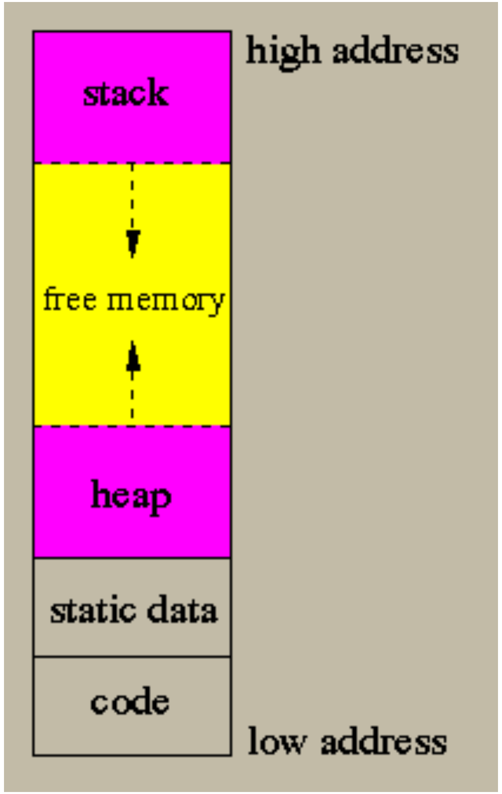


- The stack segment is where local variables reside for function calls. It’s a LIFO (Last In, First Out) data structure.

Memory Basics: Stack

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

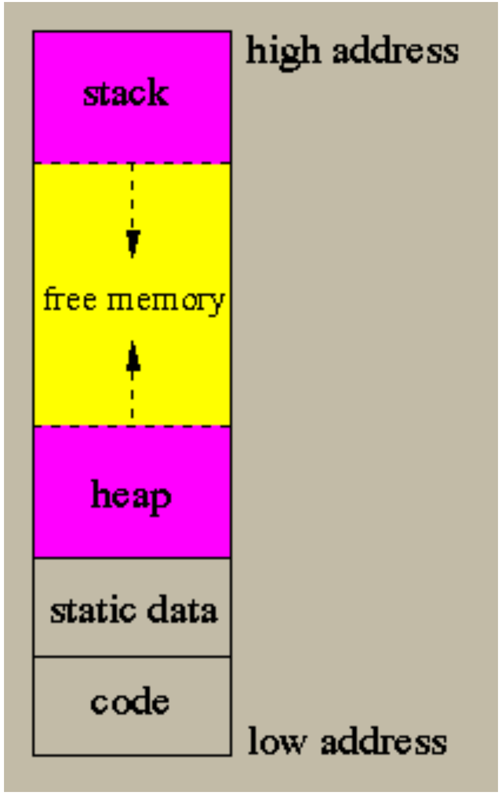
- The stack segment is where local variables reside for function calls. It's a LIFO (Last In, First Out) data structure.
- The stack is incremented by adding **stack frames**, which are contiguous blocks of memory that contain local variables for a single procedure call.



Memory Basics: Stack

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

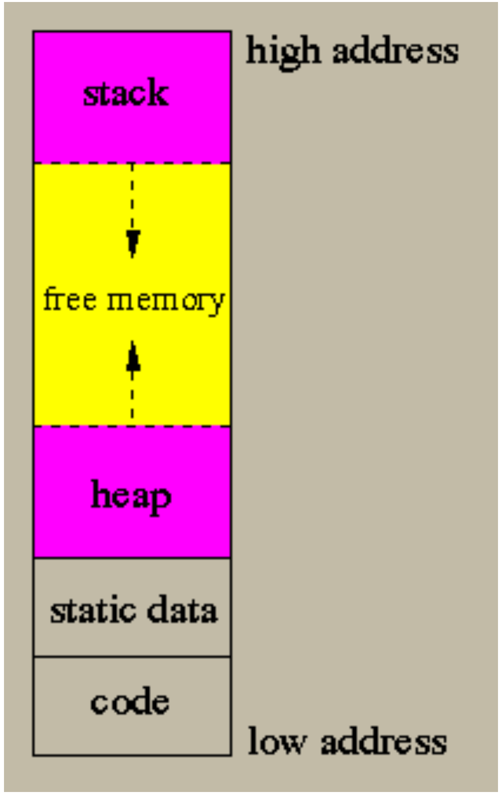
- The stack segment is where local variables reside for function calls. It's a LIFO (Last In, First Out) data structure.
- The stack is incremented by adding **stack frames**, which are contiguous blocks of memory that contain local variables for a single procedure call.
- Each stack frame contains space for the location of the calling function, its arguments, and space for local variables.



Memory Basics: Stack

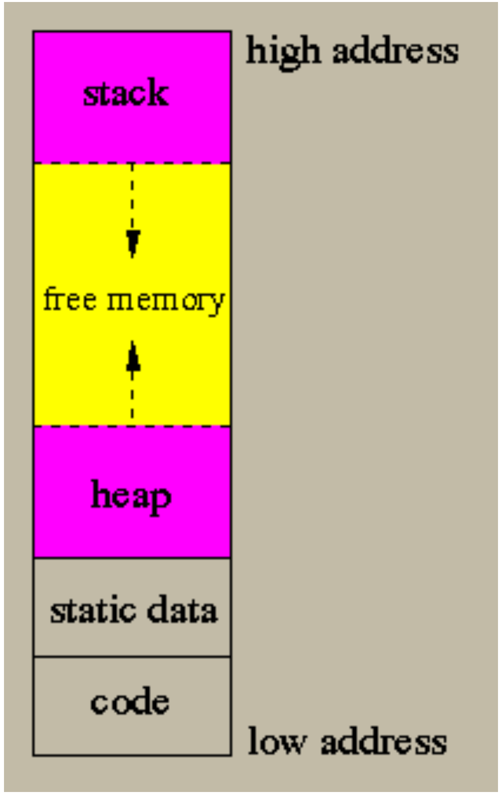
Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html

- The stack segment is where local variables reside for function calls. It's a LIFO (Last In, First Out) data structure.
- The stack is incremented by adding **stack frames**, which are contiguous blocks of memory that contain local variables for a single procedure call.
- Each stack frame contains space for the location of the calling function, its arguments, and space for local variables.
- A stack frame contains a **return address**. When the function returns, the stack pointer jumps to the return address and the memory occupied by the stack frame is automatically freed.



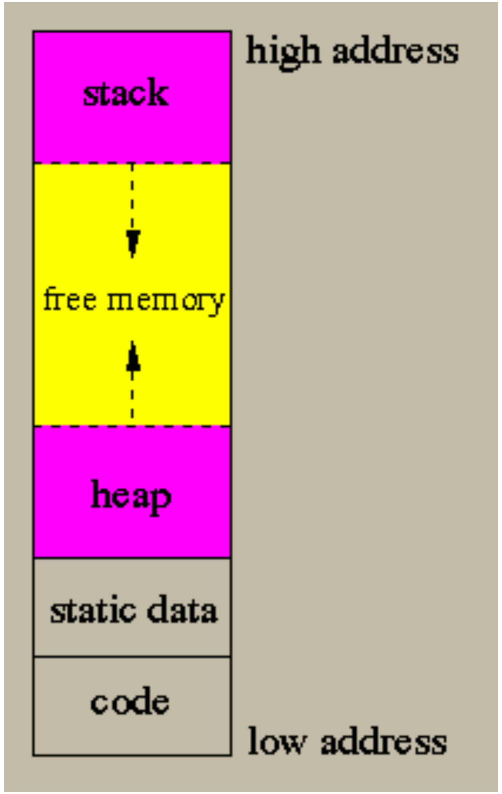
Memory Basics: Stack

Image taken from: http://lambda.uta.edu/cse5317/notes/node33.html



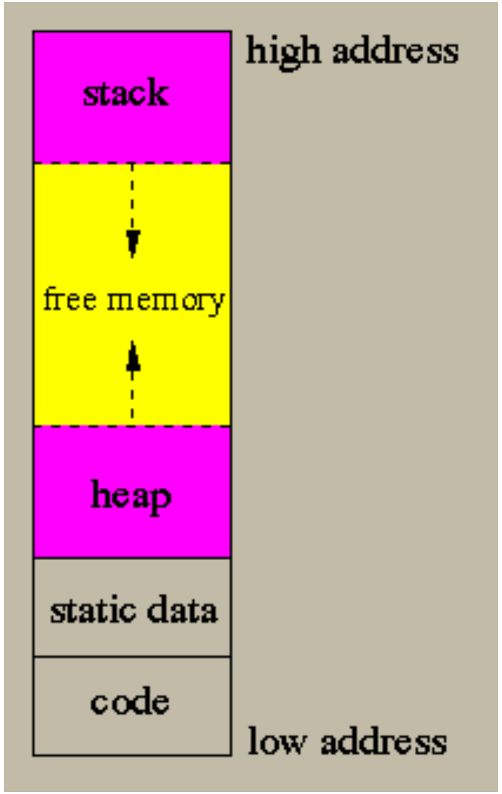
- The stack segment is where local variables reside for function calls. It's a LIFO (Last In, First Out) data structure.
- The stack is incremented by adding **stack frames**, which are contiguous blocks of memory that contain local variables for a single procedure call.
- Each stack frame contains space for the location of the calling function, its arguments, and space for local variables.
- A stack frame contains a **return address**. When the function returns, the stack pointer jumps to the return address and the memory occupied by the stack frame is automatically freed.
- The current position of the stack (lowest stack frame) is pointed to by the **stack pointer**.

Memory Basics: Stack Tips



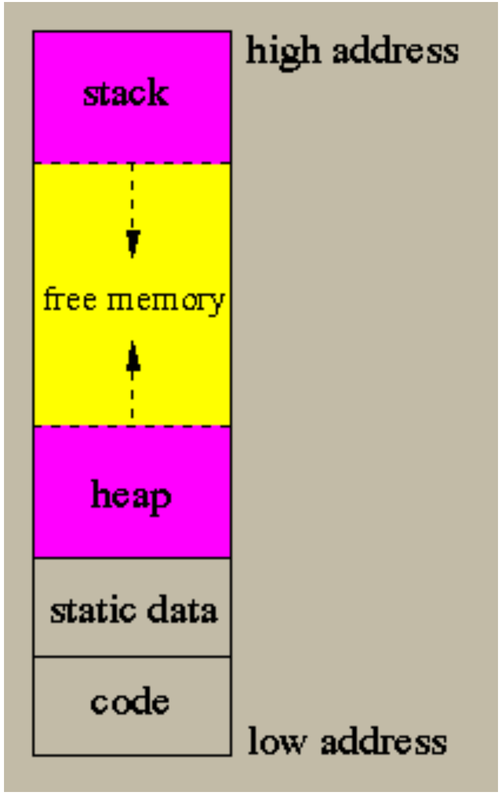
Memory Basics: Stack Tips

- Note that stack frames are freed as soon as the function they belong to returns. If you want to use things across functions, then you should allocate to the heap instead.



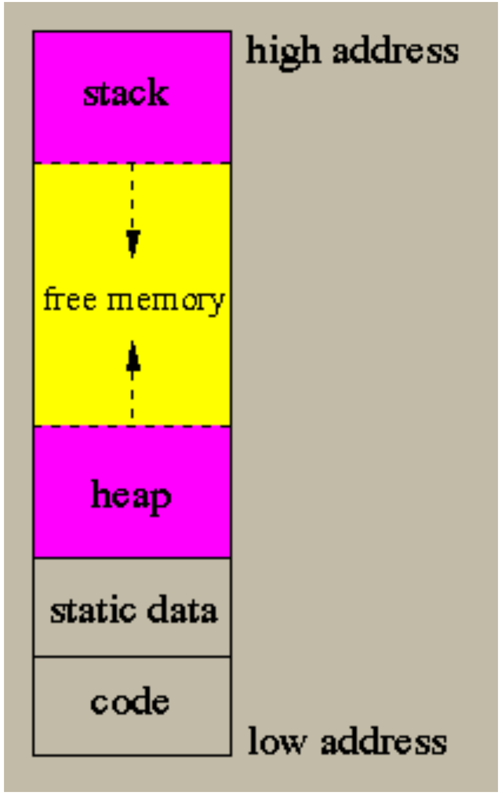
Memory Basics: Stack Tips

- Note that stack frames are freed as soon as the function they belong to returns. If you want to use things across functions, then you should allocate to the heap instead.
- For this reason you should take care never to return a pointer to a local variable. After the function returns, the pointer will be pointing to garbage.



Memory Basics: Stack Tips

- Note that stack frames are freed as soon as the function they belong to returns. If you want to use things across functions, then you should allocate to the heap instead.
- For this reason you should take care never to return a pointer to a local variable. After the function returns, the pointer will be pointing to garbage.
- A stack overflow occurs when the stack pointer collides with the heap. If too much data is allocated locally by functions, either due to excessive recursion or very large local variables, stack overflow (and a resulting segmentation fault) can occur. You can avoid this by dynamically allocating large variables and converting recursive code into iterative code (loops).



Memory Allocation

Memory Allocation

- `void *malloc(size_t size)`

Attempts to allocate ‘size’ bytes of memory on the heap and returns a pointer to the beginning of the block if successful.

Memory Allocation

- `void *malloc(size_t size)`

Attempts to allocate ‘size’ bytes of memory on the heap and returns a pointer to the beginning of the block if successful.

- `void *calloc(size_t nitems, size_t size)`

Attempts to allocate ‘nitems’ * ‘size’ bytes (nitems, size bytes each), and initializes them all to 0.

Memory Allocation

- `void *malloc(size_t size)`

Attempts to allocate ‘size’ bytes of memory on the heap and returns a pointer to the beginning of the block if successful.

- `void *calloc(size_t nitems, size_t size)`

Attempts to allocate ‘nitems’ * ‘size’ bytes (nitems, size bytes each), and initializes them all to 0.

- `void *realloc(void *ptr, size_t size)`

Attempts to change the block of memory pointed to by ‘ptr’ to be ‘size’ bytes.

Declaring Arrays

Arrays in C are contiguous blocks of memory.
They do not know their own length, unlike Java arrays.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int main(int argc, char* argv[]) {
  int A[SIZE];
  int B[] = {1,2,3,4};
  int* C = malloc(4*sizeof(int));
  if(!C){
    printf("malloc failed");
    exit(1);
  }
  free(C);
  return 0;
}
```

Declaring Arrays

Arrays in C are contiguous blocks of memory.
They do not know their own length, unlike Java arrays.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int main(int argc, char* argv[]) {
  int A[SIZE];                // Declare an array of ints of size SIZE
  int B[] = {1,2,3,4};
  int* C = malloc(4*sizeof(int));
  if(!C){
    printf("malloc failed");
    exit(1);
  }
  free(C);
  return 0;
}
```

Declaring Arrays

Arrays in C are contiguous blocks of memory.
They do not know their own length, unlike Java arrays.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int main(int argc, char* argv[]) {
  int A[SIZE];                // Declare an array of ints of size SIZE
  int B[] = {1,2,3,4}; //Declare an array of ints with initial values
  int* C = malloc(4*sizeof(int));
  if(!C){
    printf("malloc failed");
    exit(1);
  }
  free(C);
  return 0;
}
```

Declaring Arrays

Arrays in C are contiguous blocks of memory.
They do not know their own length, unlike Java arrays.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int main(int argc, char* argv[]) {
  int A[SIZE];              // Declare an array of ints of size SIZE
  int B[] = {1,2,3,4}; //Declare an array of ints with initial values
  int* C = malloc(4*sizeof(int)); // Allocate enough memory for 4 ints on heap
  if(!C){
    printf("malloc failed");
    exit(1);
  }
  free(C);
  return 0;
}
```

Declaring Arrays

Arrays in C are contiguous blocks of memory.
They do not know their own length, unlike Java arrays.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int main(int argc, char* argv[]) {
  int A[SIZE];                // Declare an array of ints of size SIZE
  int B[] = {1,2,3,4}; //Declare an array of ints with initial values
  int* C = malloc(4*sizeof(int)); // Allocate enough memory for 4 ints on heap
  if(!C){  // Check if malloc succeeds
    printf("malloc failed");
    exit(1);
  }
  free(C);
  return 0;
}
```

Declaring Arrays

Arrays in C are contiguous blocks of memory.
They do not know their own length, unlike Java arrays.

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int main(int argc, char* argv[]) {
  int A[SIZE];                 // Declare an array of ints of size SIZE
  int B[] = {1,2,3,4}; //Declare an array of ints with initial values
  int* C = malloc(4*sizeof(int)); // Allocate enough memory for 4 ints on heap
  if(!C){  // Check if malloc succeeds
    printf("malloc failed");
    exit(1);
  }
  free(C);  // Free allocated memory
  return 0;
}
```

Structs and Typedefs

- Why use structs and typedefs?

Structs and Typedefs

- Why use structs and typedefs?
- Easy way to define new data structures; structs are data structures that are composed of simpler data types.

Structs and Typedefs

- Why use structs and typedefs?
- Easy way to define new data structures; structs are data structures that are composed of simpler data types.
- Similar to classes in Java/C++, but without inheritance or methods.

Structs and Typedefs

- Why use structs and typedefs?
- Easy way to define new data structures; structs are data structures that are composed of simpler data types.
- Similar to classes in Java/C++, but without inheritance or methods.
- Typedefs are often useful to differentiate between incompatible or different things that can have the same basic type. An example is differentiating between a player's score and his ID, which may both be integers. A function that takes one should not take the other.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
struct idCard {
  unsigned int id;
  char[32] name;
};
```

```c
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

struct idCard {

  unsigned int id;

  char* name;

};
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
struct idCard {
  unsigned int id;
  char* name;
};
typedef struct idCard idCard_t;
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
typedef struct idCard {
  unsigned int id;
  char* name;
} idCard_t; // Combines struct definition with typedef
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
typedef struct idCard {
  unsigned int id;
  char* name;
} idCard_t; // Combines struct definition with typedef
void setName(idCard_t *id, char* name) {
     char* tmp = (char*) realloc(id->name,
                sizeof(char) *(strlen(name) + 1));
     if (!tmp) { //check if realloc succeeds
          printf("Realloc failed!\n");
          exit(1);
     }
     id->name = tmp;
     strcpy(id->name, name); //copy contents of name
                                     //to id->name
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
typedef struct idCard {
  unsigned int id;
  char* name;
} idCard_t; // Combines struct definition with typedef
void setName(idCard_t *id, char* name) {
     char* tmp = (char*) realloc(id->name,
                sizeof(char) *(strlen(name) + 1));
     if (!tmp) { //check if realloc succeeds
          printf("Realloc failed!\n");
          exit(1);
     }
     id->name = tmp;
     strcpy(id->name, name); //copy contents of name
                                //to id->name
}
```

```c
int main() {
  idCard_t myCard;
  myCard.id = 1001;
  setName(&myCard, "Alice");
  printf("myCard is (%u, %s)\n",
          myCard.id,myCard.name);
  return 0;
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
typedef struct idCard {
  unsigned int id;
  char* name;
} idCard_t; // Combines struct definition with typedef
void setName(idCard_t *id, char* name) {
     char* tmp = (char*) realloc(id->name,
                   sizeof(char) *(strlen(name) + 1));
     if (!tmp) { //check if realloc succeeds
          printf("Realloc failed!\n");
          exit(1);
     }
     id->name = tmp;
     strcpy(id->name, name); //copy contents of name
                             //to id->name
}
```

```c
int main() {
  idCard_t myCard;
  myCard.id = 1001;
  setName(&myCard, "Alice");
  printf("myCard is (%u, %s)\n",
          myCard.id,myCard.name);
  return 0;
}
```

Running produces:

```
myCard is (1001, Alice)
```

Enums

```
enum direction { NORTH, WEST, SOUTH, EAST } ;
typedef enum direction direction_t;
```

Enums

```
enum direction { NORTH, WEST, SOUTH, EAST } ;
typedef enum direction direction_t;
direction_t getOppositeDirection(direction_t direction) {
  switch(direction) {
    case NORTH: return SOUTH;
    case SOUTH: return NORTH;
    case EAST: return WEST;
    case WEST: return EAST;
  }
}
int main() {
  printf("Opposite of NORTH: %d", getOppositeDirection(NORTH));
  return 0;
}
```

Prints:

Enums

```
enum direction { NORTH, WEST, SOUTH, EAST } ;
typedef enum direction direction_t;
direction_t getOppositeDirection(direction_t direction) {
  switch(direction) {
    case NORTH: return SOUTH;
    case SOUTH: return NORTH;
    case EAST: return WEST;
    case WEST: return EAST;
  }
}
int main() {
  printf("Opposite of NORTH: %d", getOppositeDirection(NORTH));
  return 0;
}
```

Prints: `Opposite of NORTH: 2`

Function Pointers

A function pointer, instead of pointing to data values, points to code that is executable in memory. When a function pointer is dereferenced, it can be used to call the function that it points to, just like any other function call. This is known as an indirect call.

Function Pointers

A function pointer, instead of pointing to data values, points to code that is executable in memory. When a function pointer is dereferenced, it can be used to call the function that it points to, just like any other function call. This is known as an indirect call.

Let's write the **`map`** function.

Function Pointers

A function pointer, instead of pointing to data values, points to code that is executable in memory. When a function pointer is dereferenced, it can be used to call the function that it points to, just like any other function call. This is known as an indirect call.

Let’s write the **`map`** function.

In Python,

`map(lambda x: x*x, [1, 2, 3, 4])` returns **`[1, 4, 9, 16]`**

Function Pointers

A function pointer, instead of pointing to data values, points to code that is executable in memory. When a function pointer is dereferenced, it can be used to call the function that it points to, just like any other function call. This is known as an indirect call.

Let's write the **`map`** function.

In Python,

`map([1, 2, 3, 4], lambda x: x*x)` returns **`[1, 4, 9, 16]`**

How might we write this in C?

How would we pass a function to another function?

Function Pointers

```
#include <stdlib.h>
#include <stdio.h>
int* map(int* input, size_t length, int(*func)(int)) {
  int* newArray;
  int i;
  if (!(newArray = malloc(length*sizeof(int)))){
    printf("Malloc Failed\n");
    exit(1);
  }
  for(i = 0; i < length; i++)
    newArray[i] = func(input[i]);
  return newArray;
}
```

Function Pointers

```
#include <stdlib.h>
#include <stdio.h>
int* map(int* input, size_t length, int(*func)(int)) {
  int* newArray;
  int i;
  if (!(newArray = malloc(length*sizeof(int)))){
    printf("Malloc Failed\n");
    exit(1);
  }
  for(i = 0; i < length; i++)
    newArray[i] = func(input[i]);
  return newArray;
}
```

Function Pointers

```
#include <stdlib.h>
#include <stdio.h>
int* map(int* input, size_t length, int(*func)(int)) {
  int* newArray;
  int i;
  if (!(newArray = malloc(length*sizeof(int)))){
    printf("Malloc Failed\n");
    exit(1);
  }
  for(i = 0; i < length; i++)
    newArray[i] = func(input[i]);
  return newArray;
}
int squared(int x) {
  return x * x;
}
```

Function Pointers

```
#include <stdlib.h>
#include <stdio.h>
int* map(int* input, size_t length, int(*func)(int)) {
  int* newArray;
  int i;
  if (!(newArray = malloc(length*sizeof(int)))){
    printf("Malloc Failed\n");
    exit(1);
  }
  for(i = 0; i < length; i++)
    newArray[i] = func(input[i]);
  return newArray;
}
int squared(int x) {
  return x * x;
}
```

```
int main(){
  int array[] = {1, 2, 3, 4};
  int i;
  int* array_squared = map(array, 4,
           &squared);
  for(i = 0; i < 4; i++)
    printf("array_squared[%d]: %d\n", i,
           array_squared[i]);
  return 0;
}
```

Keywords in C

Examples of keywords: extern, const, static, if, continue, break.

Keywords in C

Examples of keywords: extern, const, static, if, continue, break.

- `extern`: declares the variable as global so that it can be used by other programs. This is the default for variables and functions at the global level. You still need the variable initialization in the source file or the linked source file.

Keywords in C

Examples of keywords: extern, const, static, if, continue, break.

- `extern`: declares the variable as global so that it can be used by other programs. This is the default for variables and functions at the global level. You still need the variable initialization in the source file or the linked source file.

```
extern int var;
int var = 10;
```

Keywords in C

Examples of keywords: extern, const, static, if, continue, break.

- `extern:` declares the variable as global so that it can be used by other programs. This is the default for variables and functions at the global level. You still need the variable initialization in the source file or the linked source file.

```
extern int var;
int var = 10;
```

- `const:` declares a variable as constant or ‘read-only’. A constant variable cannot be assigned to after initialization.

Keywords in C

Examples of keywords: extern, const, static, if, continue, break.

- `extern`: declares the variable as global so that it can be used by other programs. This is the default for variables and functions at the global level. You still need the variable initialization in the source file or the linked source file.

```
extern int var;
int var = 10;
```

- `const`: declares a variable as constant or 'read-only'. A constant variable cannot be assigned to after initialization.

```
const int var = 5;
var = 10; // error
```

Keywords in C

Examples of keywords: extern, const, static, if, continue, break.

- `extern`: declares the variable as global so that it can be used by other programs. This is the default for variables and functions at the global level. You still need the variable initialization in the source file or the linked source file.

```
extern int var;
int var = 10;
```

- `const`: declares a variable as constant or ‘read-only’. A constant variable cannot be assigned to after initialization.

```
const int var = 5;
var = 10; // error
```

- `static`: declares a variable as only visible to the file it is in (opposite of `extern`). You can also declare static variables inside a function to make that variable keep state between invocations. However, this is discouraged since it is not thread-safe and confusing.

Keywords in C

Examples of keywords: extern, const, static, if, continue, break.

- `extern`: declares the variable as global so that it can be used by other programs. This is the default for variables and functions at the global level. You still need the variable initialization in the source file or the linked source file.

```
extern int var;
int var = 10;
```

- `const`: declares a variable as constant or 'read-only'. A constant variable cannot be assigned to after initialization.

```
const int var = 5;
var = 10; // error
```

- `static`: declares a variable as only visible to the file it is in (opposite of `extern`). You can also declare static variables inside a function to make that variable keep state between invocations. However, this is discouraged since it is confusing and not thread-safe.

Full list of keywords in ANSI C available here: http://tigcc.ticalc.org/doc/keywords.html

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to “link” them when compiling.

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to “link” them when compiling.

./c_lib/functions.h

```
int rand_int();
```

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to “link” them when compiling.

./c_lib/functions.h

```c
int rand_int();
```

./functions.c

```c
#include "functions.h"

int rand_int() {
  return 4;
}
```

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to “link” them when compiling.

./c_lib/functions.h

```
int rand_int();
```

./functions.c

```
#include “functions.h”

int rand_int() {
  return 4;
}
```

./main.c

```
#include “functions.h”
#include <stdio.h>
int main() {
  printf(“%d\n”, rand_int());
  printf(“%d\n”, rand_int());
  return 0;
}
```

Header Files

- **#include <file.h>**: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to "link" them when compiling.

./c_lib/functions.h

```c
int rand_int();
```

To compile these files we need to tell gcc where to find the .c and .h files.

./functions.c

```c
#include "functions.h"

int rand_int() {
  return 4;
}
```

./main.c

```c
#include "functions.h"
#include <stdio.h>
int main() {
  printf("%d\n", rand_int());
  printf("%d\n", rand_int());
  return 0;
}
```

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to “link” them when compiling.

./c_lib/functions.h

```
int rand_int();
```

./functions.c

```
#include "functions.h"

int rand_int() {
  return 4;
}
```

./main.c

```
#include "functions.h"
#include <stdio.h>
int main() {
  printf("%d\n", rand_int());
  printf("%d\n", rand_int());
  return 0;
}
```

To compile these files we need to tell gcc where to find the .c and .h files.

```
gcc -Wall -g -I ./c_lib functions.c main.c -o main
```

Header Files

- **#include <file.h>**: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to "link" them when compiling.

./c_lib/functions.h

```c
int rand_int();
```

./functions.c

```c
#include "functions.h"

int rand_int() {
  return 4;
}
```

./main.c

```c
#include "functions.h"
#include <stdio.h>
int main() {
  printf("%d\n", rand_int());
  printf("%d\n", rand_int());
  return 0;
}
```

To compile these files we need to tell gcc where to find the .c and .h files.

```
gcc -Wall -g -I ./c_lib functions.c main.c -o main
```

Show all warnings

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to “link” them when compiling.

./c_lib/functions.h

```c
int rand_int();
```

./functions.c

```c
#include "functions.h"

int rand_int() {
  return 4;
}
```

./main.c

```c
#include "functions.h"
#include <stdio.h>
int main() {
  printf("%d\n", rand_int());
  printf("%d\n", rand_int());
  return 0;
}
```

To compile these files we need to tell gcc where to find the .c and .h files.

```
gcc -Wall -g -I ./c_lib functions.c main.c -o main
```

(-g) Create gdb symbols

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to "link" them when compiling.

./c_lib/functions.h

```
int rand_int();
```

./functions.c

```
#include "functions.h"

int rand_int() {
  return 4;
}
```

./main.c

```
#include "functions.h"
#include <stdio.h>
int main() {
  printf("%d\n", rand_int());
  printf("%d\n", rand_int());
  return 0;
}
```

To compile these files we need to tell gcc where to find the .c and .h files.

```
gcc -Wall -g -I ./c_lib functions.c main.c -o main
```

Where to find header files (points to `-I ./c_lib`)

Header Files

- `#include <file.h>`: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to "link" them when compiling.

./c_lib/functions.h

```
int rand_int();
```

./functions.c

```
#include "functions.h"

int rand_int() {
  return 4;
}
```

./main.c

```
#include "functions.h"
#include <stdio.h>
int main() {
  printf("%d\n", rand_int());
  printf("%d\n", rand_int());
  return 0;
}
```

To compile these files we need to tell gcc where to find the .c and .h files.

```
gcc -Wall -g -I ./c_lib functions.c main.c -o main
```

Where to find source files

Header Files

- **#include <file.h>**: Takes the contents of file.h and inserts it at the location of #include before compilation.
- The header files should include the variable declarations required for file.c.
- If you create your own c header files you need to "link" them when compiling.

./c_lib/functions.h

```c
int rand_int();
```

./functions.c

```c
#include "functions.h"

int rand_int() {
  return 4;
}
```

./main.c

```c
#include "functions.h"
#include <stdio.h>
int main() {
  printf("%d\n", rand_int());
  printf("%d\n", rand_int());
  return 0;
}
```

To compile these files we need to tell gcc where to find the .c and .h files.

```
gcc -Wall -g -I ./c_lib functions.c main.c -o main
```

The output file

Makefiles

Makefiles

- Nice tutorial here: http://mrbook.org/tutorials/make/

Makefiles

- Nice tutorial here: http://mrbook.org/tutorials/make/
- Running **`make`** from the command line will look for a file named **`Makefile`** in the working directory and execute it.

Makefiles

- Nice tutorial here: http://mrbook.org/tutorials/make/
- Running `make` from the command line will look for a file named `Makefile` in the working directory and execute it.
- At its most basic level, a makefile is simply composed of:

```
target: dependencies
[tab] system command
```

Makefiles

- Nice tutorial here: http://mrbook.org/tutorials/make/
- Running **`make`** from the command line will look for a file named **`Makefile`** in the working directory and execute it.
- At its most basic level, a makefile is simply composed of:

```
target: dependencies
[tab] system command
```

- As an example of the above, suppose we wanted to run:

```
gcc -g -Wall main.c hello_world.c yay.c -o hello_world
```

Makefiles

- Nice tutorial here: http://mrbook.org/tutorials/make/
- Running **`make`** from the command line will look for a file named **`Makefile`** in the working directory and execute it.
- At its most basic level, a makefile is simply composed of:

```
target: dependencies
[tab] system command
```

- As an example of the above, suppose we wanted to run:

```
gcc -g -Wall main.c hello_world.c yay.c -o hello_world
```

- We could then write this in the makefile:

```
all:
    gcc -g -Wall main.c hello_world.c yay.c -o hello_world
```

Makefiles

- Nice tutorial here: http://mrbook.org/tutorials/make/
- Running **`make`** from the command line will look for a file named **`Makefile`** in the working directory and execute it.
- At its most basic level, a makefile is simply composed of:

```
target: dependencies
[tab] system command
```

- As an example of the above, suppose we wanted to run:

```
gcc -g -Wall main.c hello_world.c yay.c -o hello_world
```

- We could then write this in the makefile:

```
all:
    gcc -g -Wall main.c hello_world.c yay.c -o hello_world
```

- The target for the above makefile is *`all`*. This is the default target for a makefile, if no other is provided. Other targets can also often be useful, since if we modify particular files in our program, we can recompile only those files instead of recompiling the entire program.

Makefiles

Makefiles

- You can put comments and variables in makefiles. Anything on a line following the `#` character is a comment. Variables are assigned with a single `=`, and you can use a variable *`VARNAME`* by calling *`$(VARNAME)`* like the following:

Makefiles

- You can put variables and comments in makefiles. Anything on a line following the `#` character is a comment. Variables are assigned with a single `=`, and you can use a variable *VARNAME* by calling `$(VARNAME)` like the following:

```
CC = gcc
all:
    $(CC) -g -Wall main.c hello_world.c yay.c -o hello_world
```

Makefiles

- You can put variables and comments in makefiles. Anything on a line following the `#` character is a comment. Variables are assigned with a single `=`, and you can use a variable *VARNAME* by calling *$(VARNAME)* like the following:

```
CC = gcc
all:
    $(CC) -g -Wall main.c hello_world.c yay.c -o hello_world
```

- A common target is `clean`, which usually is written as a system command that will clean the output files and executables created by compilation so that a “clean” compilation can be made afterward. For example:

Makefiles

- You can put variables and comments in makefiles. Anything on a line following the `#` character is a comment. Variables are assigned with a single `=`, and you can use a variable `VARNAME` by calling `$(VARNAME)` like the following:

```
CC = gcc
all:
	$(CC) -g -Wall main.c hello_world.c yay.c -o hello_world
```

- A common target is `clean`, which usually is written as a system command that will clean the output files and executables created by compilation so that a "clean" compilation can be made afterward. For example:

```
all:
	$(CC) -g -Wall main.c hello_world.c yay.c -o hello_world
#this command can be invoked by typing 'make clean'
clean:
	rm -rf *.o hello_world
```

Makefile Example

```
CC = gcc
ifeq ($(shell sw_vers 2>/dev/null | grep Mac | awk '{ print $$2}'),Mac)
    CFLAGS = -std=c99 -g -DGL_GLEXT_PROTOTYPES -I./include/ -I/usr/X11/include \
    -DOSX
    LDFLAGS = -framework GLUT -framework OpenGL \
    -L"/System/Library/Frameworks/OpenGL.framework/Libraries" \
    -lGL -lGLU -lm -lstdc
else
    CFLAGS = -std=c99  -g -DGL_GLEXT_PROTOTYPES -Iglut-3.7.6-bin
    LDFLAGS = -lglut -lGLU
endif

RM = /bin/rm -f
all: main
main: raytracer.o
    $(CC) $(CFLAGS) -o myprog raytracer.o $(LDFLAGS)
raytracer.o: raytracer.c
    $(CC) $(CFLAGS) -c raytracer.c -o raytracer.o
clean:
    $(RM) *.o myprog
```

GDB - GNU Debugger

GDB - GNU Debugger

- gdb is used to debug c files

GDB - GNU Debugger

- gdb is used to debug c files
- To use gdb you must compile with -g with gcc

GDB - GNU Debugger

- gdb is used to debug c files
- To use gdb you must compile with -g with gcc
- gdb supports:

  - breakpoints

GDB - GNU Debugger

- gdb is used to debug c files
- To use gdb you must compile with -g with gcc
- gdb supports:

  - breakpoints

  - error traceback inspection

GDB - GNU Debugger

- gdb is used to debug c files
- To use gdb you must compile with -g with gcc
- gdb supports:

  - breakpoints

  - error traceback inspection

  - stepping through the program

GDB - GNU Debugger

```
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

GDB - GNU Debugger

```
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
collin@cirrus:~/c_test$ gdb a.out
```

GDB - GNU Debugger

```
collin@cirrus:~/c_test$ gdb a.out
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Reading symbols from /home/collin/c_test/a.out...done.
(gdb)
```

GDB - GNU Debugger

```
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
collin@cirrus:~/c_test$ gdb a.out
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) run
Starting program: /home/collin/c_test/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004c4 in main () at error.c:3
3          int zero_value = *(int*)i;
(gdb)
```

GDB - GNU Debugger

```c
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
collin@cirrus:~/c_test$ gdb a.out
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) run
Starting program: /home/collin/c_test/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004c4 in main () at error.c:3
3          int zero_value = *(int*)i;
(gdb)
```

Shows the function, file, and line number of the error

GDB - GNU Debugger

```c
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
collin@cirrus:~/c_test$ gdb a.out
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) run
Starting program: /home/collin/c_test/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004c4 in main () at error.c:3
3          int zero_value = *(int*)i;
(gdb)
```

Shows the function, file, and line number of the error

Shows the code that produced the error

GDB - GNU Debugger

```
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
collin@cirrus:~/c_test$ gdb a.out
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) run
Starting program: /home/collin/c_test/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004c4 in main () at error.c:3
3         int zero_value = *(int*)i;
(gdb) print i
$1 = 0
(gdb)
```

Shows the function, file, and line number of the error

Shows the code that produced the error

GDB - GNU Debugger

```c
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
collin@cirrus:~/c_test$ gdb a.out
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) run
Starting program: /home/collin/c_test/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004c4 in main () at error.c:3
3          int zero_value = *(int*)i;
(gdb) print i
$1 = 0
(gdb) quit
```

Shows the function, file, and line number of the error

Shows the code that produced the error

GDB - GNU Debugger

```c
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
Reading symbols from /home/collin/c_test/a.out...done.
(gdb)
```

GDB - GNU Debugger

```
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) break 3
Breakpoint 1 at 0x4004c0: file error.c, line 3.
(gdb)
```

GDB - GNU Debugger

```
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) break 3
Breakpoint 1 at 0x4004c0: file error.c, line 3.
(gdb) run
Starting program: /home/collin/c_test/a.out

Breakpoint 1, main () at error.c:3
3         int zero_value = *(int*)i;
(gdb)
```

GDB - GNU Debugger

```
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) break 3
Breakpoint 1 at 0x4004c0: file error.c, line 3.
(gdb) run
Starting program: /home/collin/c_test/a.out

Breakpoint 1, main () at error.c:3
3          int zero_value = *(int*)i;
(gdb) call i = &i
$1 = 140737488348512
(gdb)
```

GDB - GNU Debugger

```
int main() {
  long i = 0;
  int zero_value = *(int*)i;
  return zero_value;
}
```

```
Reading symbols from /home/collin/c_test/a.out...done.
(gdb) break 3
Breakpoint 1 at 0x4004c0: file error.c, line 3.
(gdb) run
Starting program: /home/collin/c_test/a.out

Breakpoint 1, main () at error.c:3
3          int zero_value = *(int*)i;
(gdb) call i = &i
$1 = 140737488348512
(gdb) continue
Continuing.
[Inferior 1 (process 22438) exited with code 0140]
```

References and Credits

This presentation was possible thanks to the following references and people:

- CS61C Spring and Summer 2013 Slides and References from Dan Garcia and Justin Hsia. Links to the course webpages here: Summer 2013 and Spring 2013.

- The GNU C reference manual, website here.

- *The C Programming Language*, written by Brian Kernighan and Dennis Ritchie.

- *C Traps and Pitfalls*, written by Andrew Koenig.

- Various *man* pages and other Unix documentation.

That’s it! Any Questions?