CS212

More Pointers

Spring 2007

Announcements

- Various Parts and their due dates
- Remaining work:
  - Design document
  - Main project
- Some handy material:
  - http://computer.howstuffworks.com/c.htm
  - http://www.cygwin.com

A few reminders

- Operators:
  - **&v**: address of **v**
  - ***p**: pointer **p** and the **p**'s pointee
- Pointers:
  - L-value:
    - insert **exp** @ **p**'s pointee
    - **`*p = exp`**:
  - R-value:
    - retrieve value from **p**'s pointee
    - **`x = *p`** and **`blah(*p)`**
- Others:
  - Aliases
  - Inverse operators
  - Null pointer

Motivation Reminder

- Reminders:
  - Why are we learning about pointers?
  - Something to do with data structures and dynamic memory allocation…?
- Stack and Heap picture:

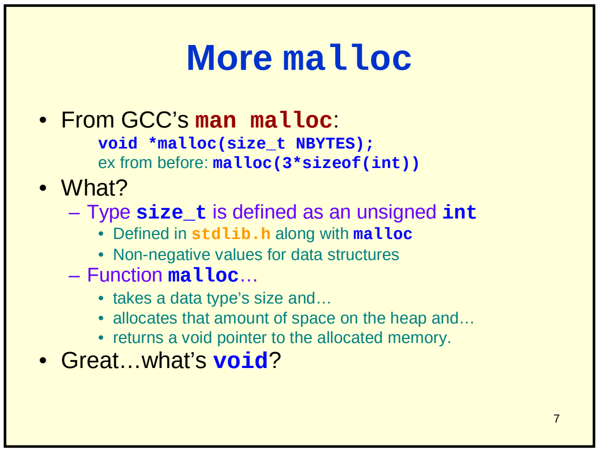
Dynamic Memory Allocation

- Memory Allocation (malloc):
  - Ex) **`malloc(3*sizeof(int))`**
  - Picture (draw below)
  - What does it return?

Visualizing Heap

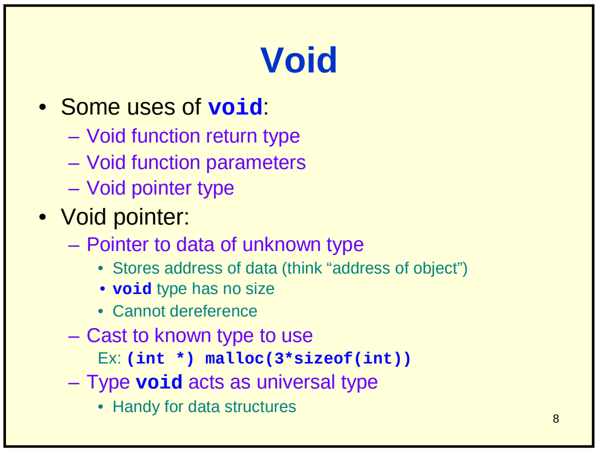
- SaM memory allocation: **`MALLOC`**!
  - pops top of Stack
  - allocates that number of cells in heap
  - pushes the address of the first cell onto stack
  - SP++
- Example (**`heap.sam`**):

```
PUSHIMM 1   // 1 cell to allocate
MALLOC      // pop 1 and allocate 1 cell in heap
PUSHIMM 3   // 3 cells to allocate
MALLOC      // pop 3 and allocate 3 cells in heap
PUSHIMM 0   // no cells to allocate
MALLOC      // pop 0 and allocate no cells in heap
FREE        // deallocate last "object"
FREE        // deallocate second "object"
FREE        // deallocate first "object"
PUSHIMM 0   // push dummy return value
STOP        // cease execution
```



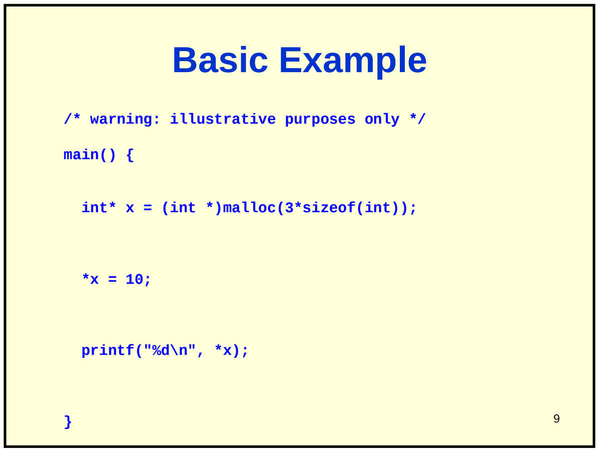
More `malloc`

- From GCC's **`man malloc`**:
  `void *malloc(size_t NBYTES);`
  ex from before: **`malloc(3*sizeof(int))`**
- What?
  - Type **`size_t`** is defined as an unsigned **`int`**
    - Defined in **`stdlib.h`** along with **`malloc`**
    - Non-negative values for data structures
  - Function **`malloc`**…
    - takes a data type's size and…
    - allocates that amount of space on the heap and…
    - returns a void pointer to the allocated memory.
- Great…what's **`void`**?



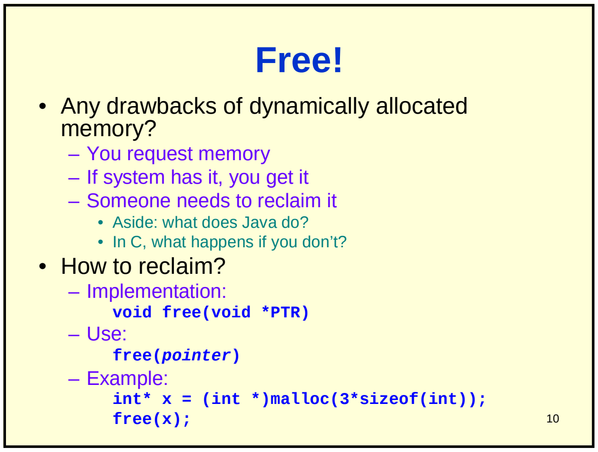
Void

- Some uses of **`void`**:
  - Void function return type
  - Void function parameters
  - Void pointer type
- Void pointer:
  - Pointer to data of unknown type
    - Stores address of data (think "address of object")
    - **`void`** type has no size
    - Cannot dereference
  - Cast to known type to use
    Ex: **`(int *) malloc(3*sizeof(int))`**
  - Type **`void`** acts as universal type
    - Handy for data structures



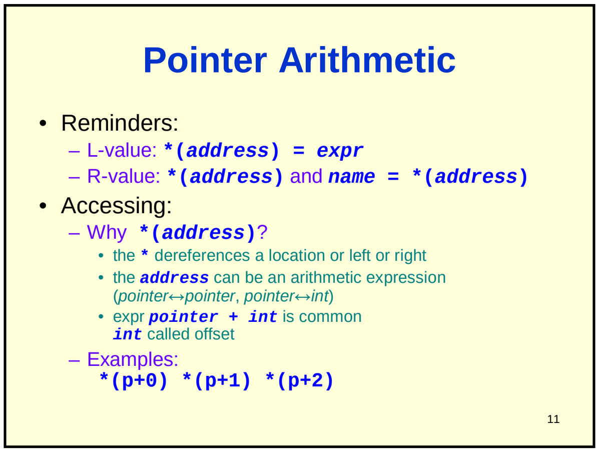
Basic Example

```
/* warning: illustrative purposes only */

main() {


  int* x = (int *)malloc(3*sizeof(int));


  *x = 10;


  printf("%d\n", *x);


}
```



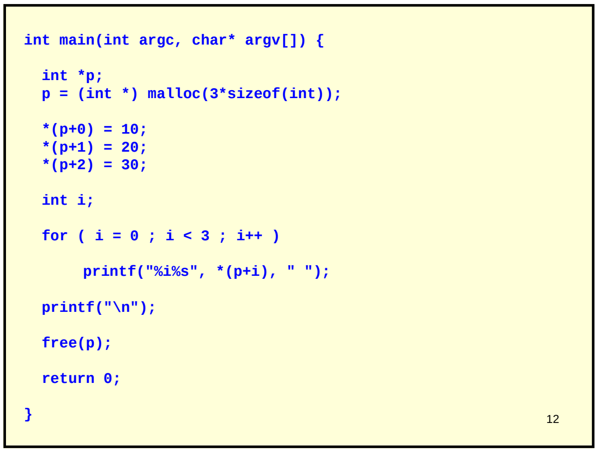
Free!

- Any drawbacks of dynamically allocated memory?
  - You request memory
  - If system has it, you get it
  - Someone needs to reclaim it
    - Aside: what does Java do?
    - In C, what happens if you don't?
- How to reclaim?
  - Implementation:
    `void free(void *PTR)`
  - Use:
    `free(`*`pointer`*`)`
  - Example:
    `int* x = (int *)malloc(3*sizeof(int));`
    `free(x);`

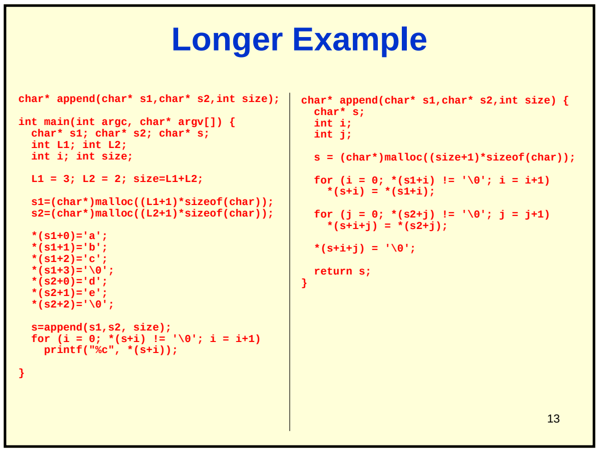


Pointer Arithmetic

- Reminders:
  - L-value: **`*(`*`address`*`) = `*`expr`***
  - R-value: **`*(`*`address`*`)`** and ***`name`* `= *(`*`address`*`)`**
- Accessing:
  - Why **`*(`*`address`*`)`**?
    - the **`*`** dereferences a location or left or right
    - the ***`address`*** can be an arithmetic expression (*pointer*↔*pointer*, *pointer*↔*int*)
    - expr ***`pointer`* `+` *`int`*** is common
      `int` called offset
  - Examples:
    `*(p+0) *(p+1) *(p+2)`



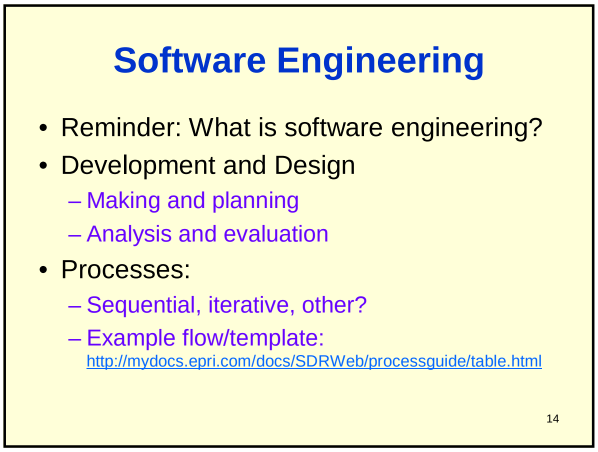
```
int main(int argc, char* argv[]) {

  int *p;
  p = (int *) malloc(3*sizeof(int));

  *(p+0) = 10;
  *(p+1) = 20;
  *(p+2) = 30;

  int i;

  for ( i = 0 ; i < 3 ; i++ )

       printf("%i%s", *(p+i), " ");

  printf("\n");

  free(p);

  return 0;

}
```



Longer Example

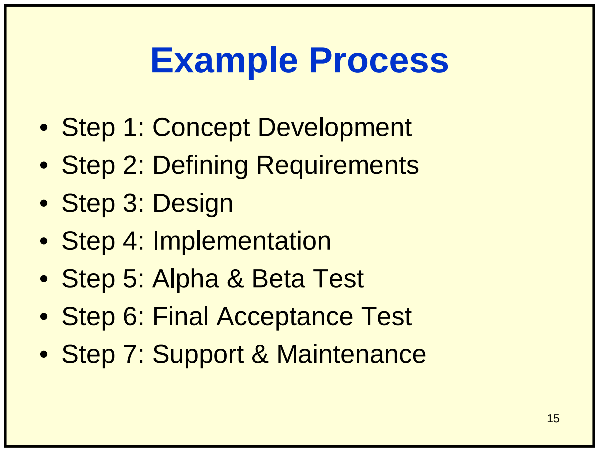
```
char* append(char* s1,char* s2,int size);

int main(int argc, char* argv[]) {
  char* s1; char* s2; char* s;
  int L1; int L2;
  int i; int size;

  L1 = 3; L2 = 2; size=L1+L2;

  s1=(char*)malloc((L1+1)*sizeof(char));
  s2=(char*)malloc((L2+1)*sizeof(char));

  *(s1+0)='a';
  *(s1+1)='b';
  *(s1+2)='c';
  *(s1+3)='\0';
  *(s2+0)='d';
  *(s2+1)='e';
  *(s2+2)='\0';

  s=append(s1,s2, size);
  for (i = 0; *(s+i) != '\0'; i = i+1)
    printf("%c", *(s+i));

}
```

```
char* append(char* s1,char* s2,int size) {
  char* s;
  int i;
  int j;

  s = (char*)malloc((size+1)*sizeof(char));

  for (i = 0; *(s1+i) != '\0'; i = i+1)
    *(s+i) = *(s1+i);

  for (j = 0; *(s2+j) != '\0'; j = j+1)
    *(s+i+j) = *(s2+j);

  *(s+i+j) = '\0';

  return s;
}
```



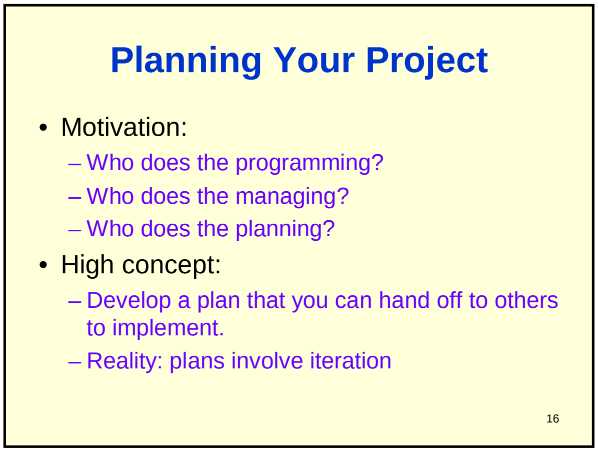
Software Engineering

- Reminder: What is software engineering?
- Development and Design
  - Making and planning
  - Analysis and evaluation
- Processes:
  - Sequential, iterative, other?
  - Example flow/template: http://mydocs.epri.com/docs/SDRWeb/processguide/table.html



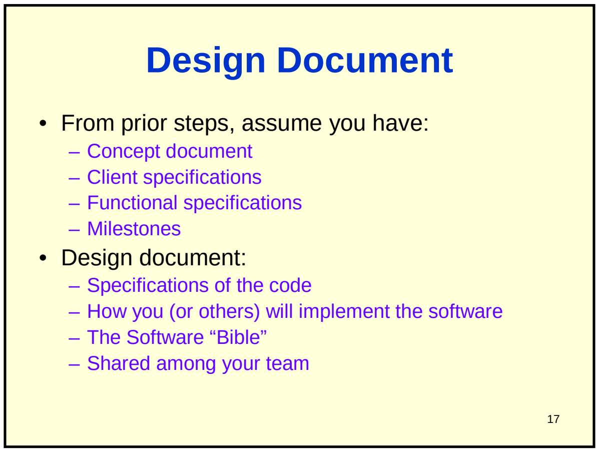
Example Process

- Step 1: Concept Development
- Step 2: Defining Requirements
- Step 3: Design
- Step 4: Implementation
- Step 5: Alpha & Beta Test
- Step 6: Final Acceptance Test
- Step 7: Support & Maintenance



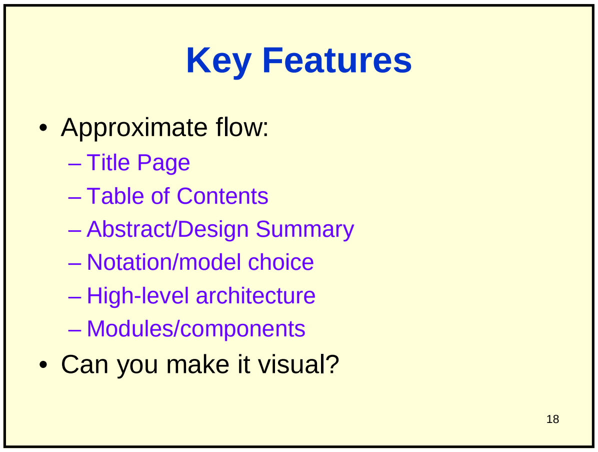
Planning Your Project

- Motivation:
  - Who does the programming?
  - Who does the managing?
  - Who does the planning?
- High concept:
  - Develop a plan that you can hand off to others to implement.
  - Reality: plans involve iteration



Design Document

- From prior steps, assume you have:
  - Concept document
  - Client specifications
  - Functional specifications
  - Milestones
- Design document:
  - Specifications of the code
  - How you (or others) will implement the software
  - The Software “Bible”
  - Shared among your team



Key Features

- Approximate flow:
  - Title Page
  - Table of Contents
  - Abstract/Design Summary
  - Notation/model choice
  - High-level architecture
  - Modules/components
- Can you make it visual?