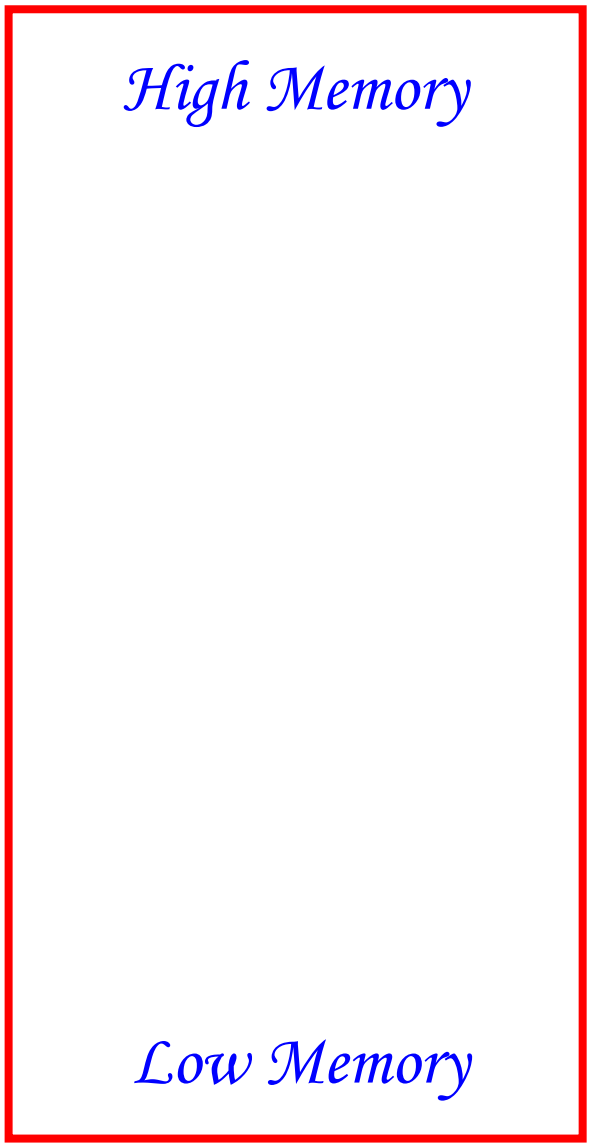
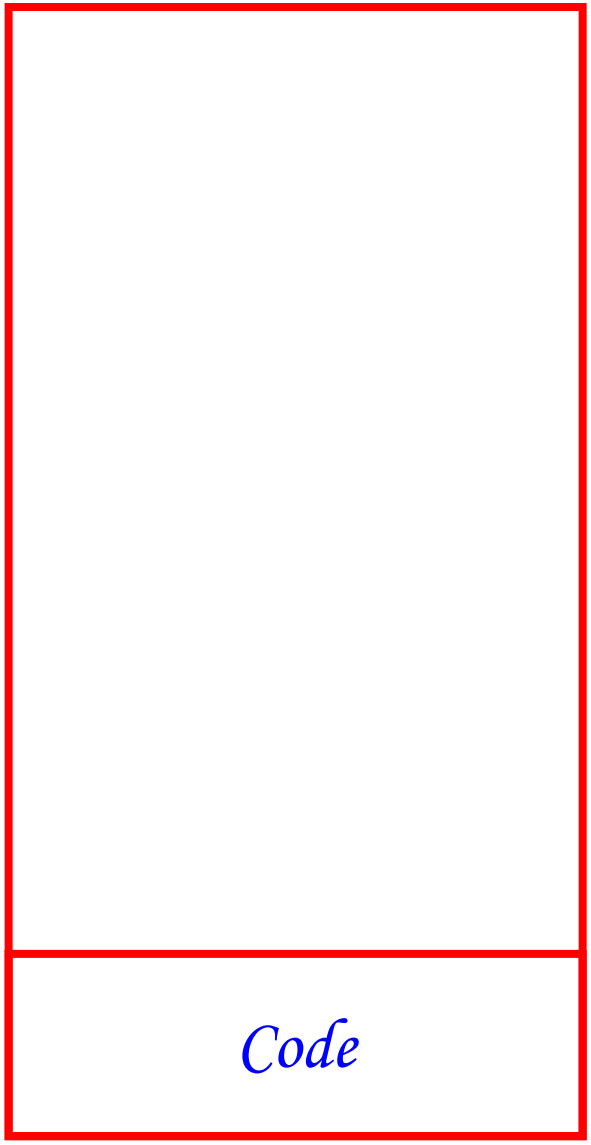
Memory Representation

- We typically draw diagrams representing the memory of the computer, our particular program or both as rectangles.
- Our convention will be that "low-memory" will be on the bottom and "high-memory" on top.
- Typically these drawings are not to scale



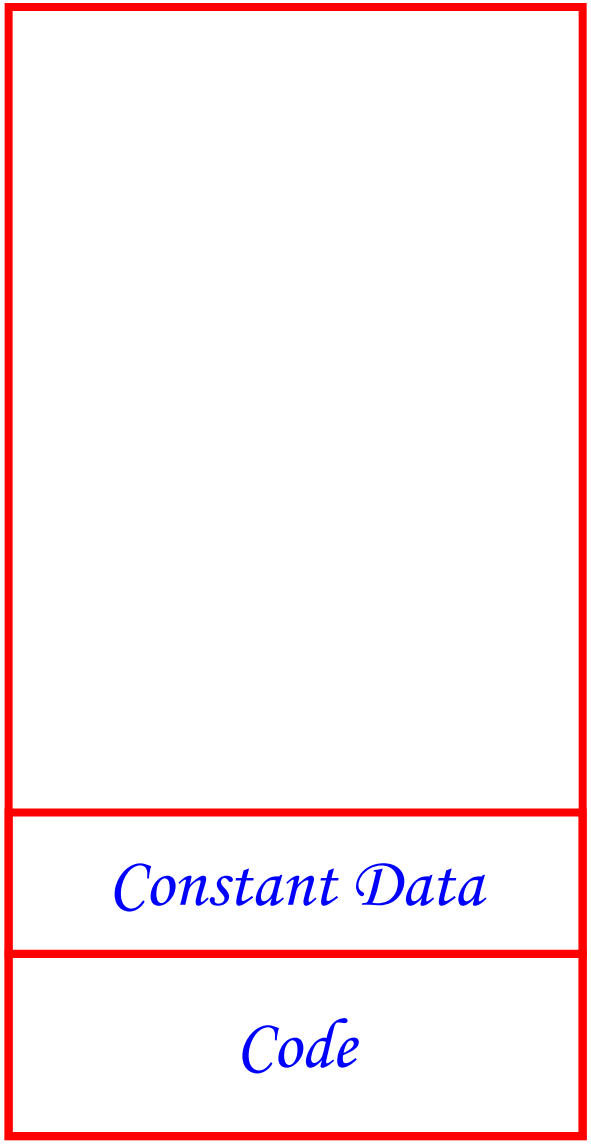
Typical Arrangement

- Normally the actual program code (executable instructions) is placed in low memory



Typical Arrangement

- Next we have an area for storage of constant data

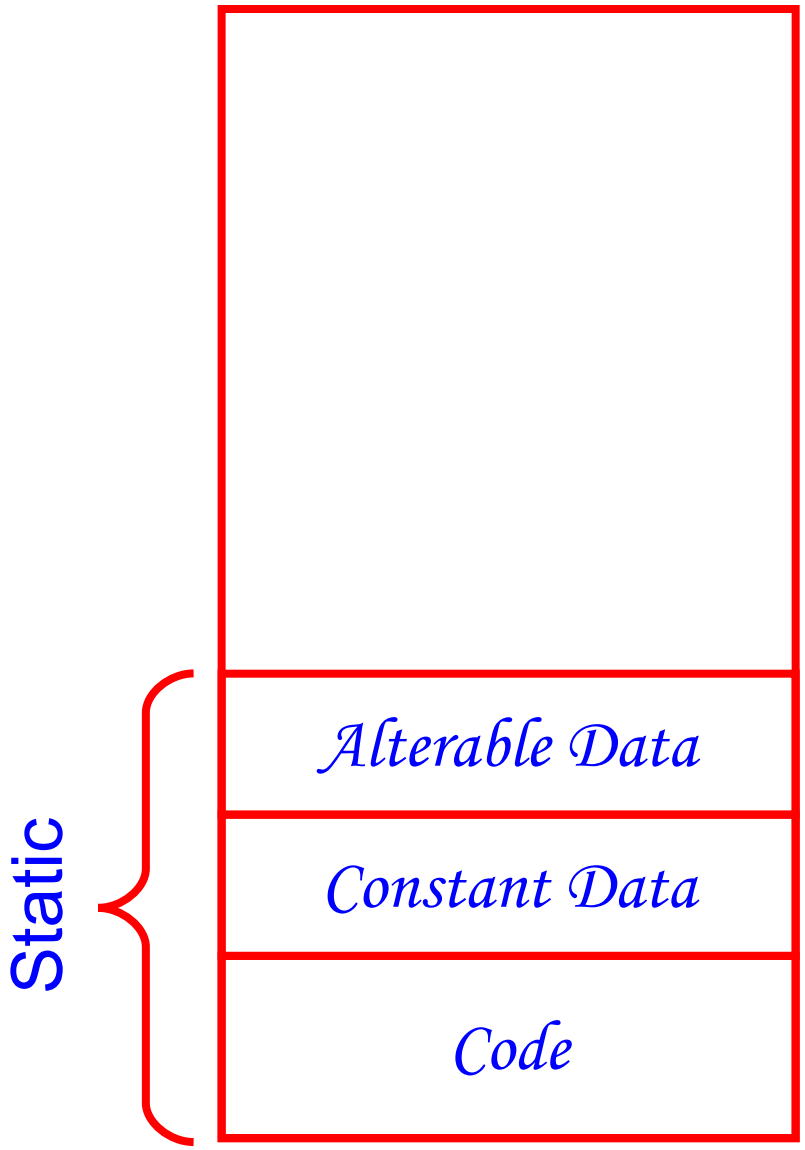


Typical Arrangement

- Data that may be changed follows

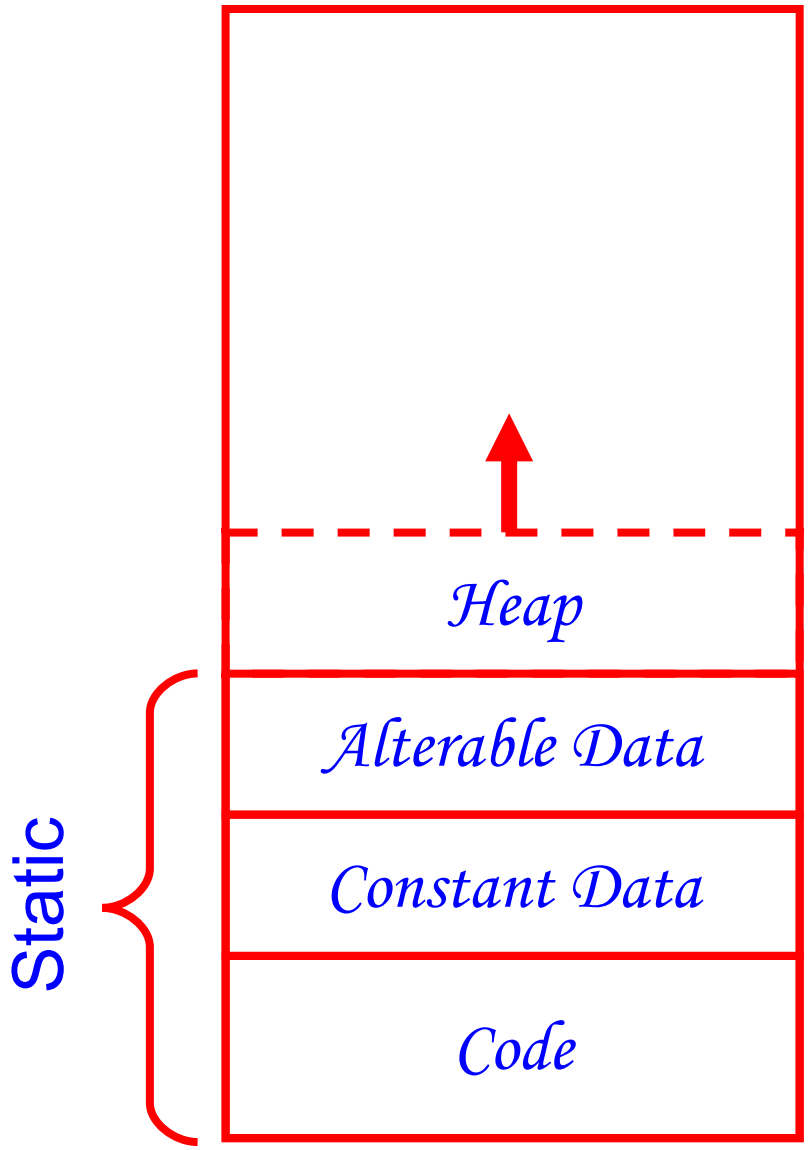
Typical Arrangement

- These three items comprise what is considered the static area of memory. The static area details (size, what is where, etc.) are known at translation or compile time.



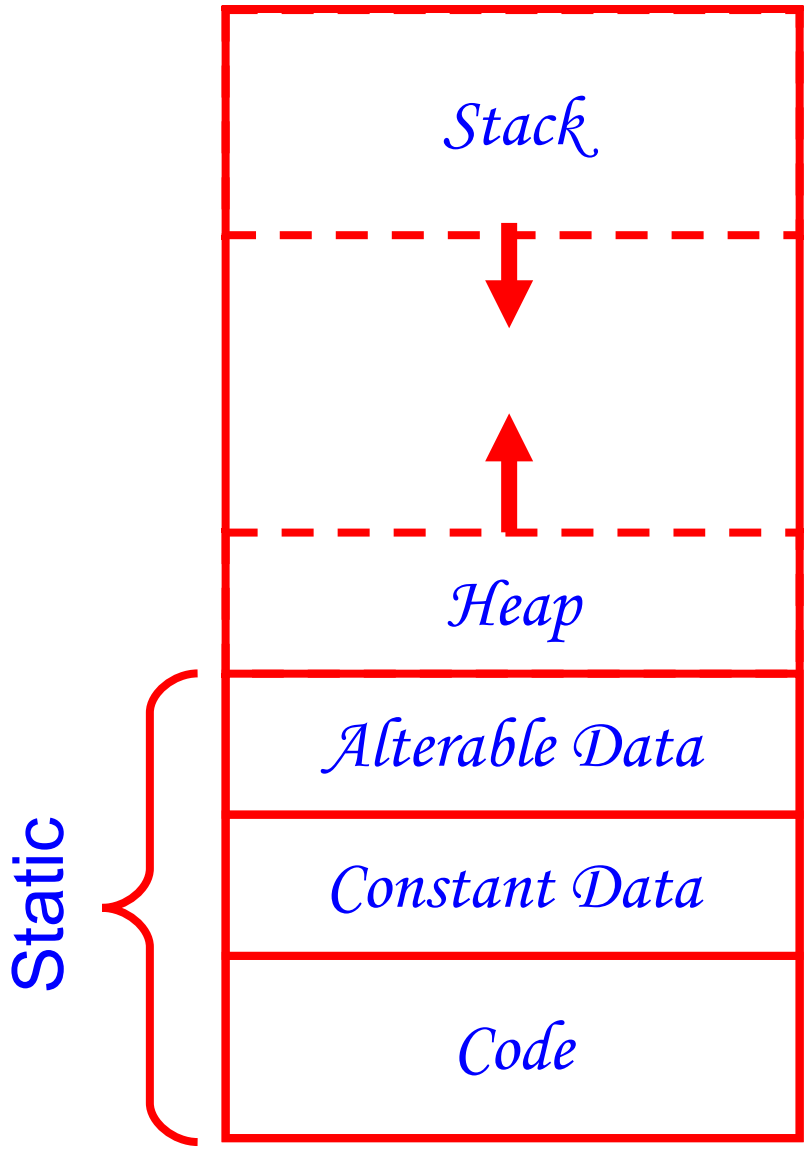
Typical Arrangement

- Immediately above the static area the heap is located.
- The heap can expand upward as the program dynamically requests additional storage space
- In most cases, the runtime environment manages the heap for the user
- We will return to this issue in a while



Typical Arrangement

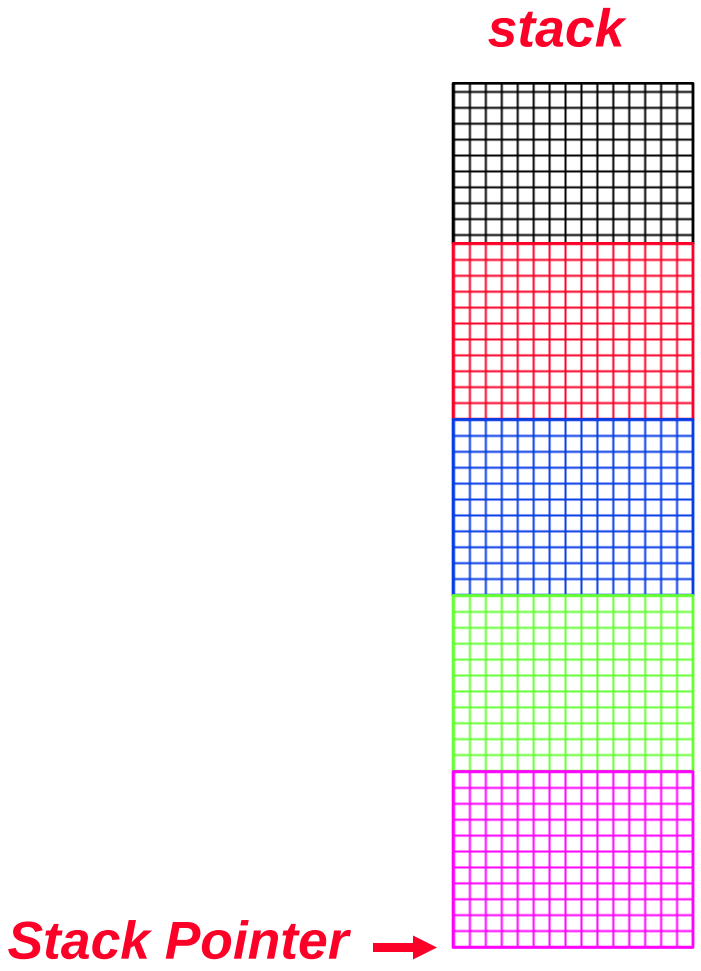
- Finally, the stack starts in high memory and can grow down as space is needed
- Stack expands with every function call and contracts with every function return
- Items maintained in the stack include
  - Local variables
  - Function parameters
  - Return values



Stack

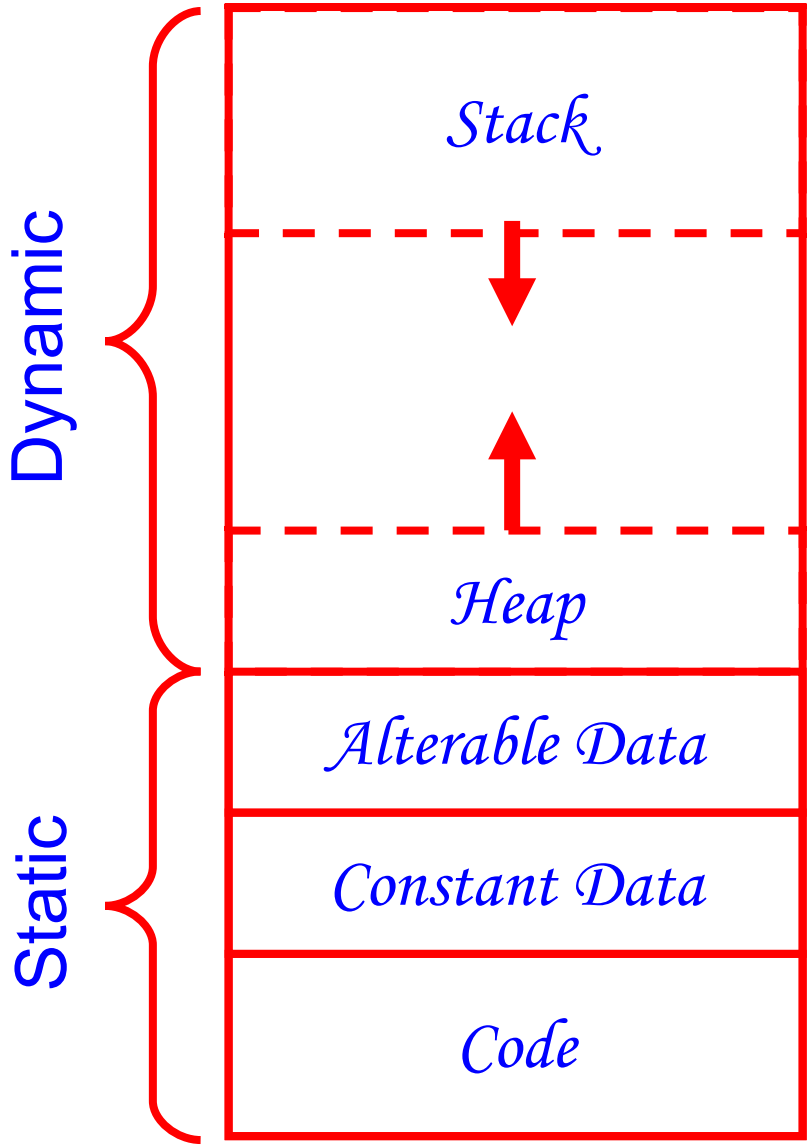
- Last In, First Out (LIFO) memory usage

```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }
        void d (int p)
        {
        }
```



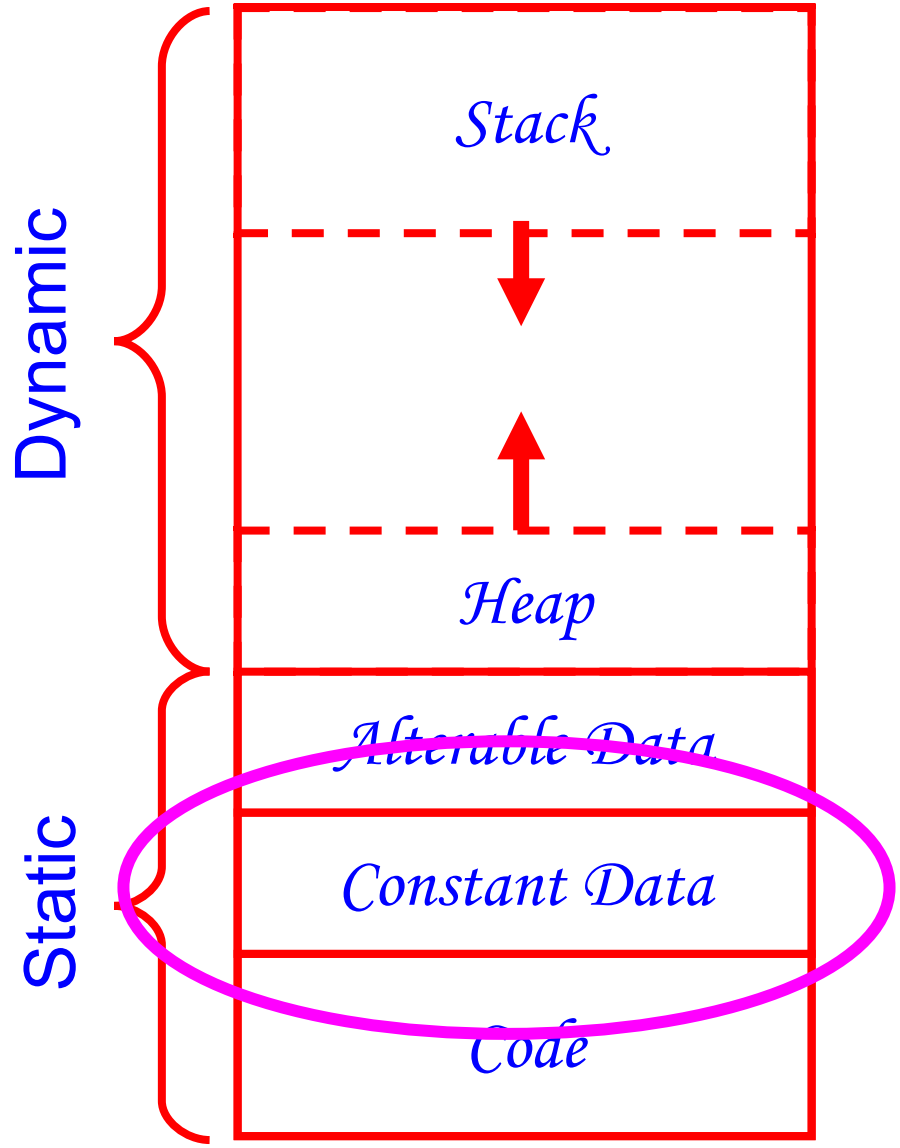
Typical Arrangement

- These items in the upper portion of the diagram change during execution of the program.
- Thus they are called dynamic



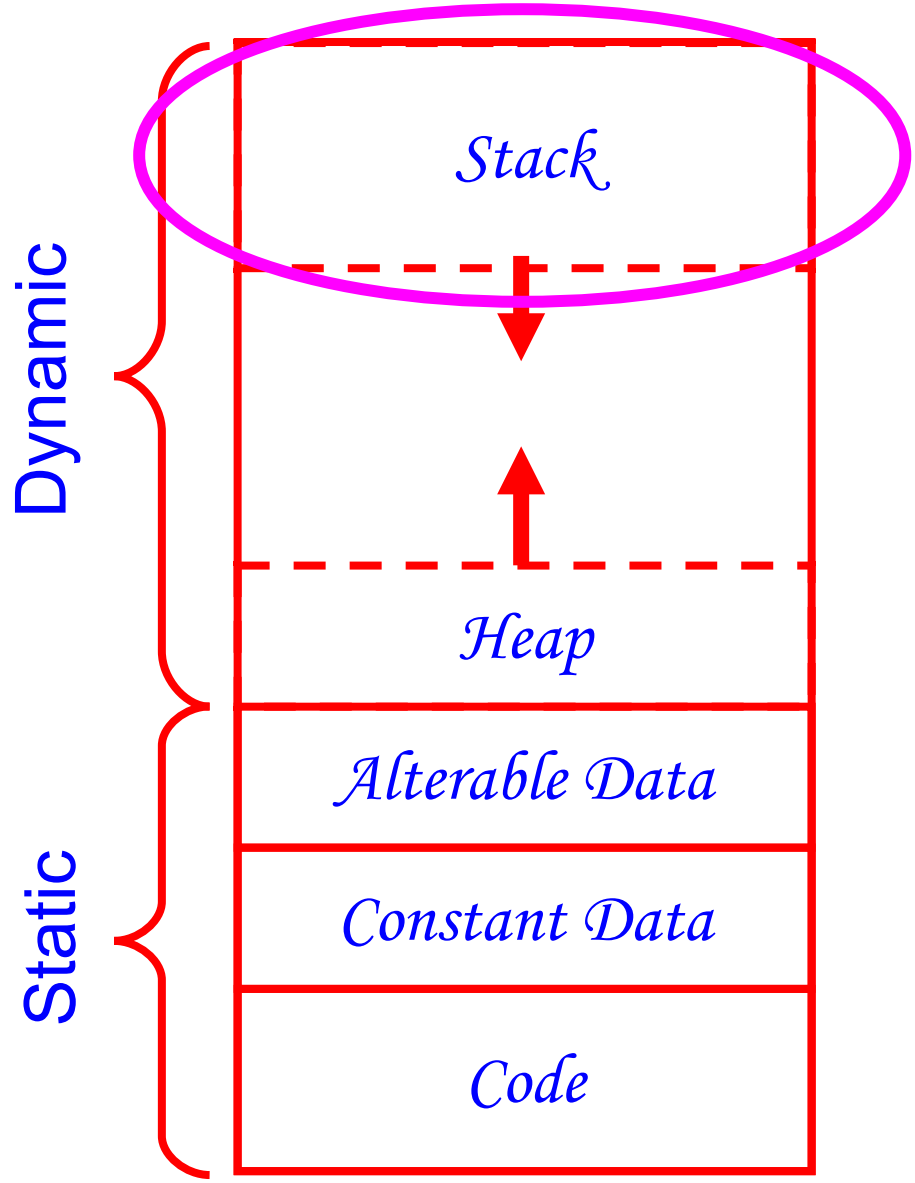
const

- Items to be maintained in the Constant Data area are designated by the programmer with the **`const`** keyword
- String constants also go there
- WARNING!!! On some systems, it is possible (and relatively easy) to modify data designated as **`const`**



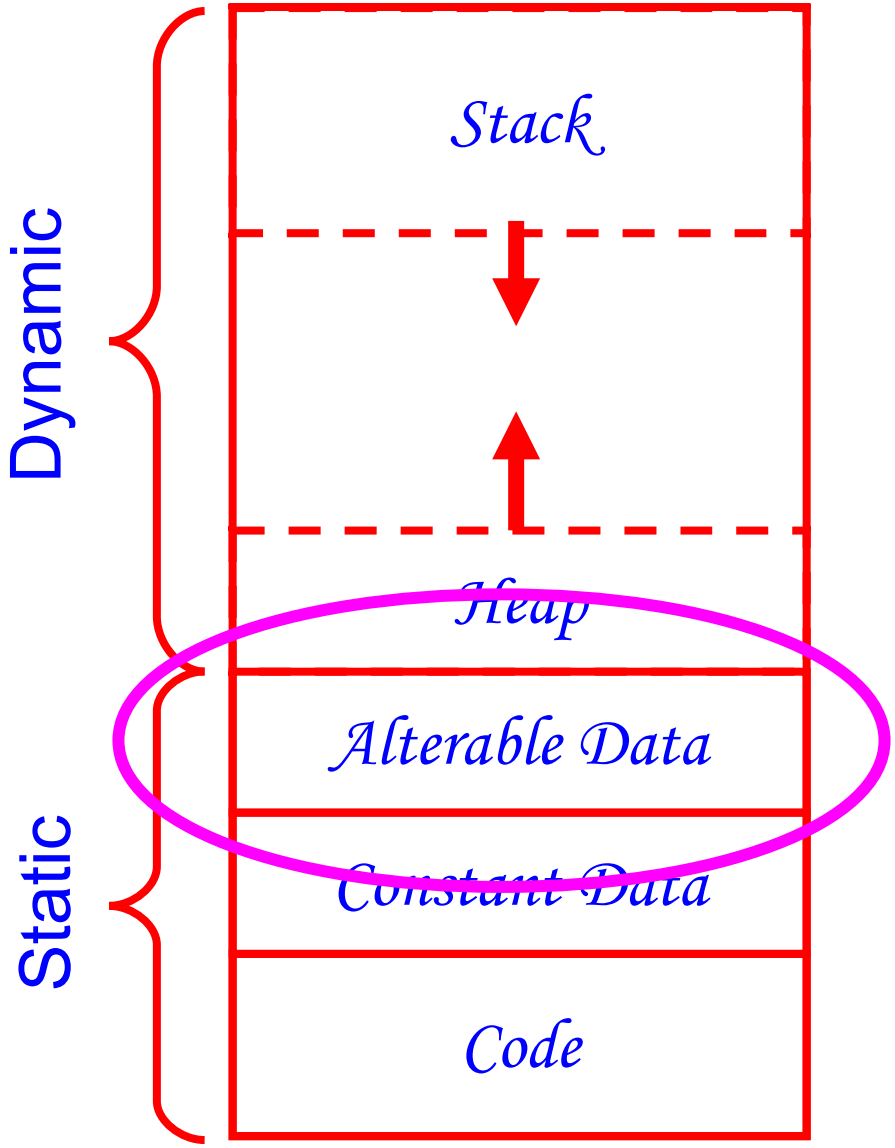
auto

- auto, short for automatic variables are those that exist on the stack. The auto keyword is not normally used.
- Automatic means that space is allocated and deallocated on the stack automatically without the programmer having to do any special operations.



`static` and `extern`

- static and extern variables exist in an area of memory set aside for alterable (or readable/writeable) data
- **`static`** can have different meanings depending on where used.



Dynamic Memory Allocation

- Fixed-sized objects, where size is known at compile-time, are stored on the stack or in the static memory area
- Sometimes you don't know the size you'll need for an array at compile-time
- You can request memory dynamically, at run time, from the *heap*
- Dynamic allocation can also be used to create memory for one object (int, structure, etc.)

Dynamic Allocation Functions

- Dynamic allocation functions:
  - **`malloc`** – allocates space that is uninitialized
  - **`calloc`** – allocates spaces that is initialized with 0's
  - **`realloc`** – re-allocates space
  - **`free`** – deallocates space
- Declared in **`<stdlib.h>`**
- Every **`malloc`** , **`calloc`** , **`realloc`** should have a matching call to **`free`**
- Otherwise, you have a *memory leak*

malloc

```
int *ip; /*  define a pointer  */
ip = malloc(10 * sizeof(int));
/*  memory for 10 elements of type int allocated */
if(ip == NULL)
{
  /* Handle Error! */
}
```

- Options for handling error
  - Abort
  - Ask again
  - Save user data
  - Ask for less
  - Free up something

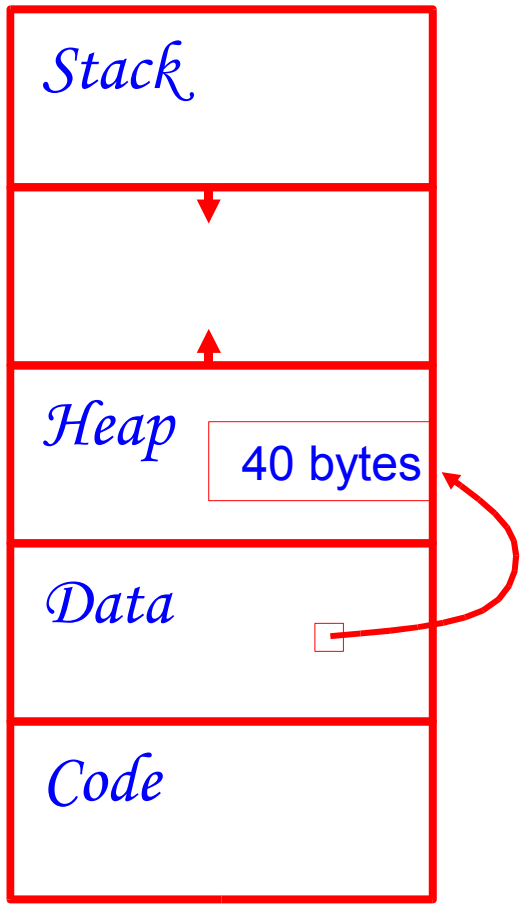
malloc

```
int *ip;
ip = malloc(10 * sizeof(int));
if(ip = NULL)
{
    /* Handle Error! */
}
```

- Note how incredibly bad it would be to use the assignment operator!
- The pointer would be set to NULL
- The error code would be skipped
- Some programmers use: `NULL == ip` or `!ip`

malloc – what happens?

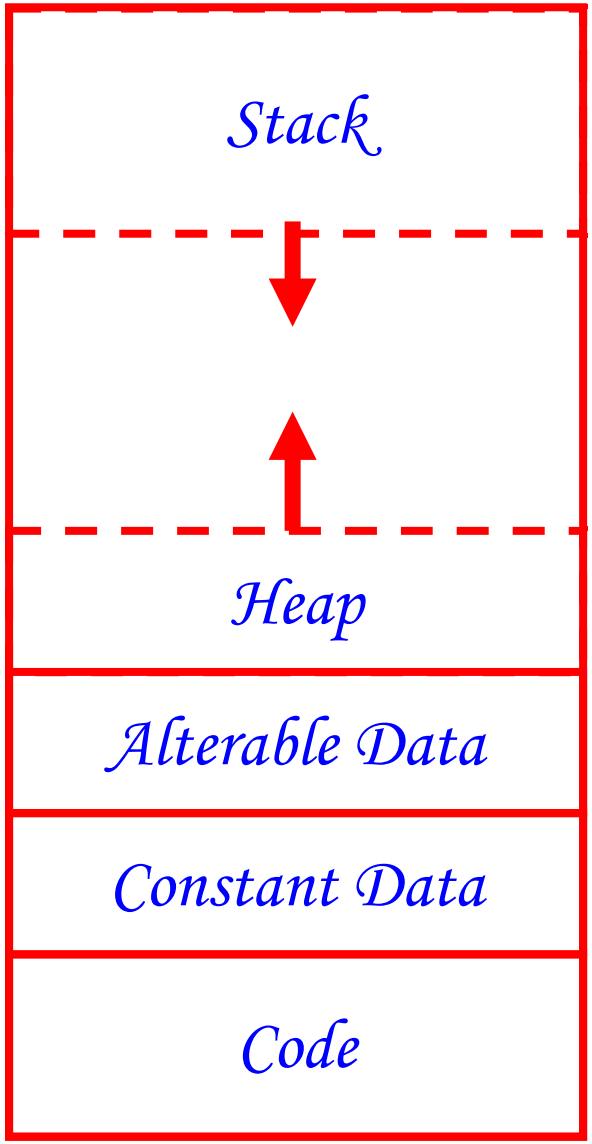
```
int *ip;
ip = malloc(10 * sizeof(int));
```



Memory Layout Example

```
#include <stdio.h>
#include <stdlib.h>

char ga[]="etext";
static char gb[]="stext";
const static char gc[]="sctext";
int i;

int
main ()
{
    static char a[]="sltext";
    static const char b[]="slctext";
    const char* p1="text";
    const char* const p2="text";
    char* p3=malloc(10);
    int j;
    printf("main=    %p\n",(void*)main);
    printf("sctext=  %p\n",(void*)gc);
    printf("slctext= %p\n",(void*)b);
    printf("p1=      %p\n",(void*)p1);
    printf("p2=      %p\n",(void*)p2);
    printf("etext=   %p\n",(void*)ga);
    printf("stext=   %p\n",(void*)gb);
    printf("sltext=  %p\n",(void*)a);
    printf("&i=      %p\n",(void*)&i);
    printf("p3=      %p\n",(void*)p3);
    printf("&p1=     %p\n",(void*)&p1);
    printf("&p2=     %p\n",(void*)&p2);
    printf("&p3=     %p\n",(void*)&p3);
    printf("&j=      %p\n",(void*)&j);
    return 0;
}
```

```
main=    0x8048394
sctext=  0x8048534
slctext= 0x804853b
p1=      0x8048543
p2=      0x8048543
etext=   0x804960c
stext=   0x8049612
sltext=  0x8049618
&i=      0x8049728
p3=      0x8049738
&p1=     0xbffff498
&p2=     0xbffff494
&p3=     0xbffff490
&j=      0xbffff48c
```



Using The Space

```
int i;
int *ip;
if((ip = malloc(10*sizeof(int))) == NULL)
{
    /* Handle Error Here */
}
for(i = 0; i < 10; i++)
    ip[i] = i;
```

Flexibility

```
#define MAX 10
int *ip;
ip = malloc(MAX * sizeof(int));
```

- What if we change the type of `int *ip`???

```
#define MAX 10
int *ip;
ip = malloc(MAX * sizeof(*ip));
```

Prototypes

- `void *malloc(size_t n);`
- `void free(void *p);`
- `void *realloc(void *p, size_t n);`
- What is this mysterious **`void`** pointer?

`void` pointer

- Not originally in C
- Relatively recent addition
- Basically a “generic” pointer
- Intended for use in applications like **`free`** where the block of memory located at some address will be freed without any necessity of defining the type

Powerful and Dangerous

```
void *vp;
char *cp;
int *ip;
ip = cp; /* illegal */
```

- Instead

```
ip = (int *)cp;
```

- or

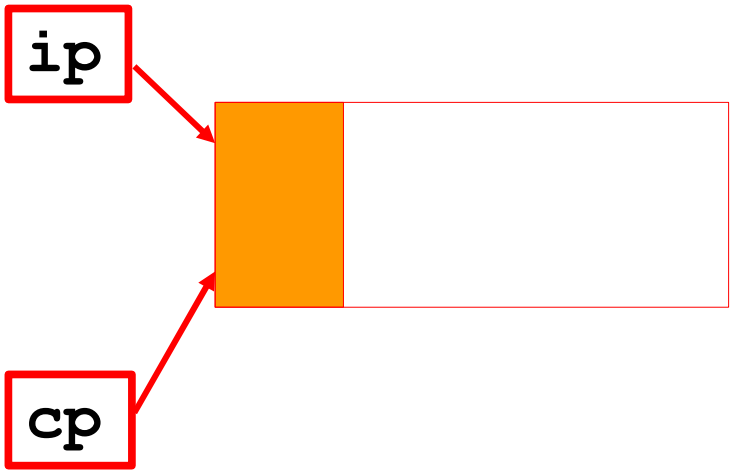
```
vp = cp;  /* Legal, powerful and */
ip = vp;  /* dangerous!!! */
```

- Why is this being done?

Casting

- Usually casting is not required
- May be masking a problem

```
int *ip;
char *cp;
...
*cp = 'x';
*ip = ???

*ip = 42;
*cp = ???
```



Warnings

- Using **`void`** pointers as a crutch to get around casting is a "bad" thing!
- **`malloc`** doesn't care what you are doing with a block of memory it allocates to you. What you do with the memory is your responsibility
- Passing in random values to free is a bad thing!
- **`free`** can change contents of block that was freed
- **`free`** does not change pointer
- After a call to **`free`** it is usually possible to do anything to the freed memory that was possible before the call!!!
- Definitely a bad thing!!!

Initializing Memory

- Use **malloc** when you do not need the memory initialized:

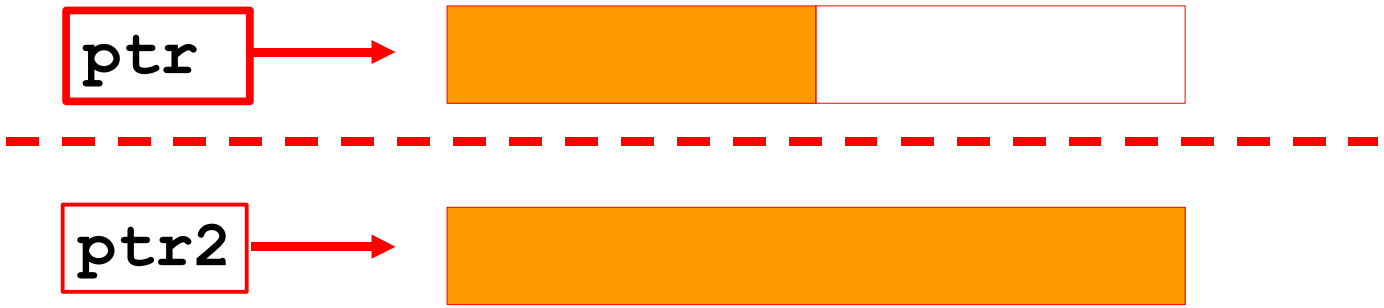
```
double *a;  /*  define a pointer  */
a = malloc(100*sizeof(double));
 /*  memory for 100 elements of type double
 allocated – they have "random" values*/
a[5] = 4.5; /*  use a as an array  */
```

- Use **calloc** when you want to initialize allocated memory:

```
double *a;  /*  define a pointer  */
a = calloc(100, sizeof(double));
 /*  memory for 100 elements of type double
 allocated and initialized with 0s*/
a[5] = 4.5; /*  use a as an array  */
```

Reallocating memory

- ptr2 = realloc(ptr, num_bytes);
- What it does (conceptually)
  - Find space for new allocation
  - Copy original data into new space
  - Free old space
  - Return pointer to new space



Dynamic Allocation

- `int *ip = malloc(...);`
- `malloc` may allocate more space than requested
- Why?
- Efficiency
- Typically if you ask for 1 byte you will get 8.
- Given this line of code
- `char *cp = malloc(1);`
- Which is more likely
  - Program will probably keep this memory as is
  - Program will eventually `realloc`
- How much can you safely use?

Safety

- Program should only use memory actually requested
- Big problem
- Program that oversteps bounds may work
- Sometimes!
- Note...

```
char *cp = malloc(1);
```

ADDR SIZE

cp 8 (maybe!)

- Now...

```
realloc(cp,6);
```

- will return same pointer thus...

realloc

- May return same pointer passed to it without indicating any problem.
- Using memory beyond that which has been allocated may work
- Some of the time
- Normally it will work when tested by a programmer but will fail when shipped to the customer

realloc

- Realloc may return
  - same pointer
  - different pointer
  - NULL
- Is this a good idea?

```
cp = realloc(cp, n);
```

- No!
- If `realloc` returns NULL `cp` is lost
- Memory Leak!

How to Do It

```
void *tmp;
if((tmp = realloc(cp,...)) == NULL)
{
    /* realloc error */
}
else
{
    cp = tmp;
}
```

Additional Information

- `realloc(NULL, n)` ≡ `malloc(n);`
- `realloc(cp, 0)` ≡ `free(cp);`
- These can be used to make **`realloc`** work in a single loop design to build a dynamic structure such as a linked list.
- Some people like to define wrappers around memory allocation functions

```
void* xmalloc(size_t size)
{
 void* ptr = malloc(size);
 if(!ptr) abort(); else return ptr;
}
```

Dynamic Stack

```
/* stack.h */
void push(int a);
int pop(void);
int peek(void);
void clear(void);
void init(void);
void finalize(void);
int empty(void);
```

```
/* stack.c */
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"
static unsigned int top;
                  /* first free slot on the stack */
static int *data;
static unsigned int size;
void init(void)
{
        top=0;
        size=0;
        data=0;
}
void finalize(void)
{
  free(data);
}
void clear(void)
{
        top=0;
}
int empty(void)
{
  return(top==0);
}
```

Dynamic Stack

```
void push(int a)
{
        if(top>=size)
        {
          unsigned int newsize=(size+1)*2;
          int* ndata=realloc(data,newsize*sizeof(int));
          if(ndata)
            data=ndata;
          else
          {
            free(data);
            abort();
          }
          fprintf(stderr,"Stack size %d -> %d\n",size,newsize);
          size=newsize;
        }
        data[top++]=a;
}
int pop(void)
{
        assert(top>0);
        return data[--top];
}
int peek(void)
{
        assert(top>0);
        return data[top-1];
}
```

size_t

- Some unsigned type
- The maximum value of variable of this type can be obtained using the expression

  `(size_t)-1`
- C99 defines the constant **`SIZE_MAX`** for that purpose

Dynamic Stack Revisited

```
void push(int a)
{
  if(top>=size)
  {
    unsigned int newsize;
    int* ndata;

    if (size == 0)
      newsize = 1;
    else if (size <= UINT_MAX/2)
      newsize = 2 * size;
    else
    {
      free(data);
      abort();
    }

    if (newsize <= ((size_t)-1) / sizeof(int))
      ndata=realloc(data,newsize*sizeof(int));
    else
    {
      free(data);
      abort();
    }

    if(ndata)
      data=ndata;
    else
    {
      free(data);
      abort();
    }
    fprintf(stderr,"Stack size %d -> %d\n",size,newsize);
    size=newsize;
  }
  data[top++]=a;
}
```

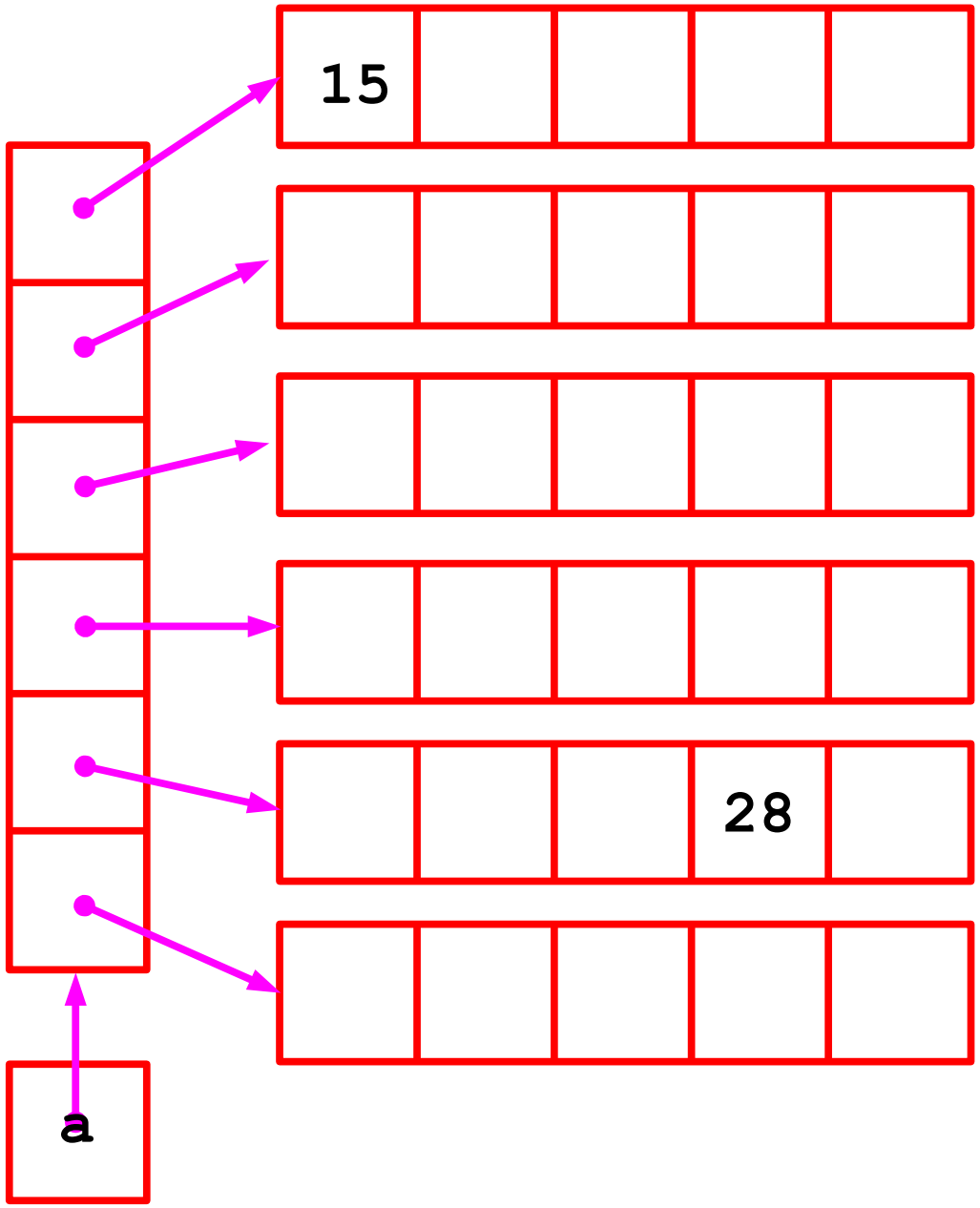
Reading Lines from Standard Input

```
char* readline()
{
  char* line = NULL;
  int c;
  size_t bufsize = 0;
  size_t size = 0;
  while((c=getchar()) != EOF)
  {
    if (size >= bufsize)
    {
      char* newbuf;
      if (bufsize == 0)
        bufsize = 2;
      else if (bufsize <= ((size_t)-1)/2)
        bufsize = 2*size;
      else
      {
        free(line);
        abort();
      }
      newbuf = realloc(line,bufsize);
      if (!newbuf)
      {
        free(line);
        abort();
      }
      line = newbuf;
    }
    line[size++]=c;
    if (c == '\n') break;
  }

  if ((c == EOF) && (size == 0))
    return NULL;
  if(size >= bufsize)
  {
      char* newbuf;
      if (size < (size_t)-1)
        bufsize = size + 1;
      else
      {
        free(line);
        abort();
      }

      newbuf = realloc(line,bufsize);
      if (!newbuf)
      {
        free(line);
        abort();
      }
      line = newbuf;
  }
  line[size++]='\0';
  return line;
}
```

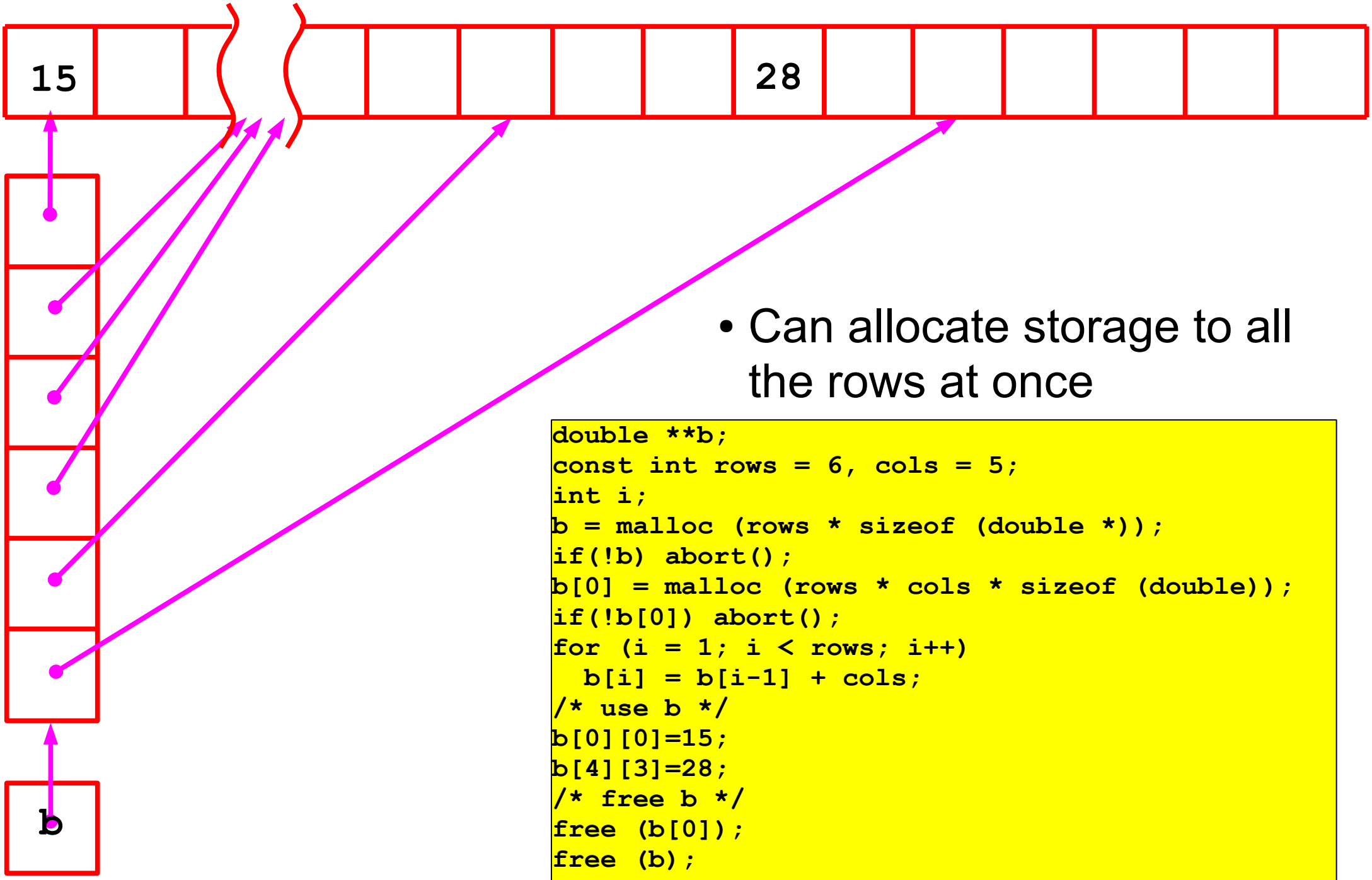
Two-Dimensional Arrays

- Pointer to a pointer
- Must first allocate the storage for the pointers to the rows, then for the contents

```
int i;
const int rows = 6, cols = 5;
double **a;
a = malloc (rows * sizeof (double *));
if(!a) abort();
for (i = 0; i < rows; i++)
{
  a[i] = malloc (cols * sizeof (double));
  if(!a[i]) abort();
}
/* use a */
a[0][0]=15;
a[4][3]=28;
/* free a */
for (i = 0; i < rows; i++)
  free (a[i]);
free (a);
```



Two-Dimensional Arrays



- Can allocate storage to all the rows at once

```
double **b;
const int rows = 6, cols = 5;
int i;
b = malloc (rows * sizeof (double *));
if(!b) abort();
b[0] = malloc (rows * cols * sizeof (double));
if(!b[0]) abort();
for (i = 1; i < rows; i++)
  b[i] = b[i-1] + cols;
/* use b */
b[0][0]=15;
b[4][3]=28;
/* free b */
free (b[0]);
free (b);
```

Printing Lines in Reverse Order

```
#include <stdio.h>
#include <stdlib.h>
#include "readline.h"


int main ()
{
  char **lines = NULL;
  size_t nolines = 0;
  size_t nolinesmax = 0;
  char *line;
  size_t i;

  while ((line = readline ()))
    {
      if (nolines >= nolinesmax)
      {
        char **newlines;
        if (nolines == 0)
          nolinesmax = 1;
        else if (nolines <= ((size_t) - 1)
                 / 2 / sizeof (char *))
        {
          nolinesmax = 2 * nolines;
        }
        else
          goto error;
        newlines = realloc (lines,
            nolinesmax * sizeof (char *));
        if (newlines == NULL)
          goto error;
        lines = newlines;
      }
      lines[nolines++] = line;
    }

  for (i = nolines; i > 0; i--)
  {
    printf ("%s", lines[i - 1]);
    free (lines[i - 1]);
  }
  free (lines);
  return 0;

error:
  for (i = nolines; i > 0; i--)
    free (lines[i - 1]);
  free (lines);
  abort ();
}
```

strdup

- **strdup** duplicates a string
- It is not a standard function, although is present in many systems
- If your system does not have it, you can define it yourself

```
char* strdup(const char* s)
{
 char* p = 0;
 p = malloc(strlen(s)+1);
 if (p)
  strcpy(p, s);
 return p;
}
```

- Why +1? Why check p? What are the “ownership semantics”?

strdup

- Calling **strdup** …

```
int main()
{
 /* Make a copy: strdup allocates memory! */
 char* copy = strdup("surgeon");

 /* Use a copy */
 printf("Like a %s\n", copy);

 /* Deallocate memory */
 free(copy);
 copy = NULL; /* So we don't accidentally use it */
 return 0;
}
```

free

- Make sure to free memory once your done with it
- Always set the variable to NULL after freeing it (Why?)

```
char* psz = strdup("Hello");
free(psz);
psz = 0;
```

- Don’t try to free the same variable twice in a row:

```
char* psz = strdup("Hello");
free(psz);
free(psz); /* boom! */
```

- Don’t try to free a variable that’s pointing to statically allocated memory:

```
char* psz = "Hello";
free(psz); /* bye bye! */
```

Memory Leaks

- Memory leaks occur when you forget to call **`free`**
- Unlike Java, C has no automatic garbage collection
- Particularly fatal to long-running processes that do many allocations (e.g. servers, daemons)
- Usually the result of
  - Simple forgetfulness
  - Multiple return paths
  - Reassigning the pointer without calling **`free`** first, esp. for in/out parameters
  - When freeing a structure, forgetting to also free the structure members
  - Not realizing when a function allocates memory that the caller is responsible for freeing

Memory Leaks

- Once a leak has been introduced, can be very hard to track down
- You need to carefully track variables that are associated with dynamic memory line by line, from birth to death
- Memory profilers helpful – ElectricFence, valgrind
- Besides memory, what other things can be leaked?

Dynamic Allocation – What Can Go Wrong

- Allocate a block of memory and use the contents without initialization
- Free a block but continue to use the contents
- Call **`realloc`** to expand a block of memory and then once moved continue to use the old address
- Allocate a block and lose it by losing the value of the pointer
- Read or write beyond the boundaries of the block
- FAIL TO NOTICE ERROR CONDITIONS

Typical Errors and `valgrind`

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);/*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep
```

Typical Errors and `valgrind`

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);/*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep

Conditional jump or move depends on uninitialised value(s)
 at 0x4027ACE2: _IO_vfprintf (in /lib/libc-2.2.5.so)
 by 0x402823B5: _IO_printf (in /lib/libc-2.2.5.so)
 by 0x8048428: main (errors.c:11)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)

Syscall param write(buf) contains uninitialised or unaddressable byte(s)
 at 0x402F2404: __libc_write (in /lib/libc-2.2.5.so)
 by 0x40298E87: (within /lib/libc-2.2.5.so)
 by 0x40298DE5: _IO_do_write (in /lib/libc-2.2.5.so)
 by 0x4029913F: _IO_file_overflow (in /lib/libc-2.2.5.so)
 Address 0x40228000 is not stack'd, malloc'd or free'd
```

Typical Errors and valgrind

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);    /*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep

Invalid write of size 1
 at 0x8048437: main (errors.c:13)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
 Address 0x41050024 is 0 bytes inside a block of size 10 free'd
 at 0x4002698D: free (vg_replace_malloc.c:231)
 by 0x8048433: main (errors.c:12)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
```

Typical Errors and valgrind

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);    /*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep

Invalid write of size 1
 at 0x8048462: main (errors.c:16)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
 Address 0x41050060 is 0 bytes inside a block of size 10 free'd
 at 0x40026C58: realloc (vg_replace_malloc.c:310)
 by 0x804845B: main (errors.c:15)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
```

Typical Errors and valgrind

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
  p1=malloc(2000000000);    /*19*/
  *p1='c';                  /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors 2>rep

Invalid write of size 1
 at 0x8048479: main (errors.c:18)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)
 Address 0x410527AC is 0 bytes after a block of size 10000 alloc'd
 at 0x40026C58: realloc (vg_replace_malloc.c:310)
 by 0x804845B: main (errors.c:15)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors)

VG_(get_memory_from_mmap): request for 2000003072 bytes failed.
VG_(get_memory_from_mmap): 14933038 bytes already allocated.

This may mean that you have run out of swap space,
since running programs on valgrind increases their memory
usage at least 3 times.  You might want to use 'top'
to determine whether you really have run out of swap.
If so, you may be able to work around it by adding a
temporary swap file -- this is easier than finding a
new swap partition.  Go ask your sysadmin(s) [politely!]

VG_(get_memory_from_mmap): out of memory!  Fatal!  Bye!
```

Typical Errors and `valgrind`

```
#include <stdio.h>          /* 1*/
#include <stdlib.h>         /* 2*/
#include <assert.h>         /* 3*/
                            /* 4*/
int                         /* 5*/
main ()                     /* 6*/
{                           /* 7*/
  char* p1, *p2;            /* 8*/
                            /* 9*/
  p1=malloc(10);            /*10*/
  printf("%c\n",p1[0]);     /*11*/
  free(p1);                 /*12*/
  *p1='a';                  /*13*/
  p1=malloc(10);            /*14*/
  p2=realloc(p1,10000);     /*15*/
  *p1='b';                  /*16*/
  malloc(30);               /*17*/
  p2[10000]='c';            /*18*/
                            /*19*/
                            /*20*/
  return 0;                 /*21*/
};                          /*22*/
```

```
$ valgrind --leak-check=yes ./errors2 2>rep

ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 2 from 1)
malloc/free: in use at exit: 10030 bytes in 2 blocks.
malloc/free: 4 allocs, 2 frees, 10050 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 2 not-freed blocks.
checked 3413768 bytes.

30 bytes in 1 blocks are definitely lost in loss record 1 of 2
 at 0x400266DE: malloc (vg_replace_malloc.c:153)
 by 0x8048470: main (errors2.c:17)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors2)

10000 bytes in 1 blocks are definitely lost in loss record 2 of 2
 at 0x40026C58: realloc (vg_replace_malloc.c:310)
 by 0x804845B: main (errors2.c:15)
 by 0x4024814E: __libc_start_main (in /lib/libc-2.2.5.so)
 by 0x8048350: (within /home/gwj/pdsc/03-lecture04/04-memerrors/errors2)

LEAK SUMMARY:
 definitely lost: 10030 bytes in 2 blocks.
 possibly lost:   0 bytes in 0 blocks.
 still reachable: 0 bytes in 0 blocks.
 suppressed: 0 bytes in 0 blocks.
 Reachable blocks (those to which a pointer was found) are not shown.
 To see them, rerun with: --show-reachable=yes
```

Structures

- A collection of member variables
- Very useful for grouping related data

```
struct Person
{
        char* name;
        int age;
};
```

Creating a Person

- Access member variables using a dot (.)

```
int main()
{
        struct Person artist;
        artist.name = strdup("Kayah");
        artist.age = 37;
       return 0;
}
```

- Can anyone spot a problem?

Another Way to Initialize A Struct

- Similar to an array, can specify an initialization list
- In C90, initialization list can contain only constants
- In C99, we could move **`strdup`** to the initializer

```
int main()
{
   struct Person artist = { NULL, 37 };
   artist.name=strdup("Kayah");
   if(!artist.name) abort();
   ... use artist ...

   /* Free memory allocated by strdup */
   free(artist.name);
   artist.name = 0;
   return 0;
}
```

Creating a Person Dynamically

- Can allocate space for a structure using **`malloc()`**
- When accessing member variables of a pointer to a structure use ->

```
int main()
{
  struct Person* artist = malloc(sizeof(struct Person));
  if(!artist) abort();
  artist->name = strdup("Kayah");
  if(!artist->name) abort();
  artist->age = 37;

  ... exploit artist ...

  /* First, free the member variables */
  free(artist->name);
  artist->name = 0; /* So we don't use it */

  /* Then, free the structure */
  free(artist);
  artist = 0; /* So we don't use it */
  return 0;
}
```

Structure typedefs

- You can make a typedef for the struct:

```
typedef struct Person SPerson;
```

- Or, similar to **enums**, often we'll combine the struct declaration with a typedef:

```
typedef struct Person {
  char* name;
  int age;
} SPerson, *SPersonPtr;
```

- Can now say **SPerson** instead of **struct Person**

Nested Structures

- Can nest structures arbitrarily

```
typedef struct Date {
  int mon, day, year;
} SDate, *SDatePtr;

typedef struct Person {
  char* name;
  SDate dob;
} SPerson, *SPersonPtr;
```

Nested Structures

- Mixture of static (.) and pointer (->) memory access

```
int
main ()
{
  SPerson *artist = malloc (sizeof (SPerson));
  if(!artist) abort();

  artist->name = strdup ("Kayah");
  if(!artist->name) abort();
  artist->dob.day = 5;
  artist->dob.mon = 11;
  artist->dob.year = 1967;

  /* Free memory allocated by strdup */
  free (artist->name);
  artist->name = 0;

  /* Free the person */
  free (artist);
  artist = 0;
  return 0;
}
```

Self-referential Structures

- A structure can have member variables that point to the same structure type

```
typedef struct Person {
  char* name;
  SDate dob;
  struct Person* parents[2];
} SPerson, *SPersonPtr;
```

Self-referential Structures

```
SPerson parents[2];
SPerson artist;

parents[0].name = strdup("Kayah's Mother");
parents[1].name = strdup("Kayah's Father");

artist.name = strdup("Kayah");
artist.parents[0] = &parents[0];
artist.parents[1] = &parents[1];

printf("%s's parents are %s and %s\n",
        artist.name, artist.parents[0]->name,
        artist.parents[1]->name);
```

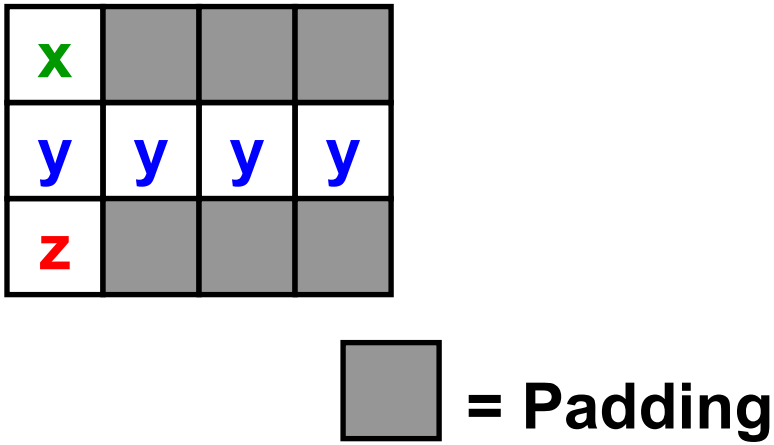
- Most frequently, dynamic memory allocation is used in such cases

Structures and sizeof

- Due to memory alignment restrictions, the size of a structure is >= the sum of the sizes of its member variables

```
struct blah {
  char x;
  int  y;
  char z;
};
```

Memory layout



- Always use sizeof to determine the size of a structure
- **`sizeof(struct blah)` ≡ 12 bytes**

Structure Bit Fields

- Recall bit flags and bit masks
- Useful when we need to pack several flags or objects into the smallest amount of space possible
- Structure bit fields make this a little easier at the cost of portability

```
struct argb {
  unsigned int alpha : 8;
  unsigned int red   : 8;
  unsigned int green : 8;
  unsigned int blue  : 8;
};
```

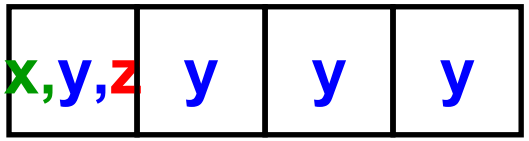
- Implementation-dependent!

Unions

- Syntactically similar to structures
- However, all member variables occupy the same location in memory
- You are responsible for accessing the right members at the right time
- Union size is size of the largest member

```
union UBlah {
  char x;
  int  y;
  char z;
};
```

Memory layout



Unions

```
union {
    char   x;
    int  y;
    char*  z;
} utype;

utype.x = 'c';
printf("%c\n", utype.x);
utype.z = "Hello";
printf("%s\n", utype.z);
printf("%d\n", utype.y); /* Undefined! */
```