Pointers and Arrays, Pointer Arithmetic, Dynamic Memory

MATH 5061: Fundamentals of Computer Programming for Scientists and Engineers

Dr. Richard Berger
richard.berger@temple.edu

Department of Mathematics
Temple University

10/27/2016

TEMPLE UNIVERSITY

Outline

Scope and Lifetime - Part 2
- Calling functions with pointer arguments
- References
- Calling functions with reference arguments

Pointers and Arrays
- Pointers of array variables

Pointer Arithmetic

Dynamic Memory Allocation

TEMPLE UNIVERSITY

Outline

Scope and Lifetime - Part 2
- Calling functions with pointer arguments
- References
- Calling functions with reference arguments

Pointers and Arrays
- Pointers of array variables

Pointer Arithmetic

Dynamic Memory Allocation

TEMPLE UNIVERSITY

Calling functions with pointer arguments

```c
#include <stdio.h>

void f(int * pa) {
    printf("f(): pa=%p, *pa=%d (before)\n", pa, *pa);
    *pa = 50;
    printf("f(): pa=%p, *pa=%d (after)\n", pa, *pa);
}

int main() {
    int a = 1;
    printf("main(): &a=%p, a=%d (before)\n", &a, a);
    f(&a);
    printf("main(): &a=%p, a=%d (after)\n", &a, a);
    return 0;
}
```

by passing the pointer `&a` as parameter `pa`, we can change the variable `a` inside the function `f()`

TEMPLE UNIVERSITY

Calling functions with pointer arguments

Output

```
main(): &a=0x7ffdfd0583ec, a=1 (before)
f(): pa=0x7ffdfd0583ec, *pa=1 (before)
f(): pa=0x7ffdfd0583ec, *pa=50 (after)
main(): &a=0x7ffdfd0583ec, a=50 (after)
```

References

```
int a = 50;
int & b = a;

printf("a=%d\n", a);
printf("b=%d\n", b);
```

Output

```
a=50
b=50
```

- a **reference** is declared using a `&` after the datatype
- you assign to it another variable
- a reference then points to that another variable
- you can then use it as an alias
- so, it acts like a pointer, but it's less typing
- no memory addresses and no dereferencing with `*`

TEMPLE UNIVERSITY

References

```
int a = 50;
int & b = a;

b = 30;

printf("a=%d\n", a);
printf("b=%d\n", b);
```

Output

```
a=30
b=30
```

- any change to the reference variable modifies the original variable

Calling functions with **reference** arguments

```
#include <stdio.h>

void f(int & b) {
    printf("f(): b=%p, b=%d (before)\n", &b, b);
    b = 50;
    printf("f(): b=%p, b=%d (after)\n", &b, b);
}

int main() {
    int a = 1;
    printf("main(): &a=%p, a=%d (before)\n", &a, a);
    f(a);
    printf("main(): &a=%p, a=%d (after)\n", &a, a);
    return 0;
}
```

- we can write the same program as with pointers before
- code becomes simpler with references

Note:
reference `b` will automatically have the same memory address as the original variable `a`

TEMPLE UNIVERSITY

Calling functions with **reference** arguments

Output

```
main(): &a=0x7ffdd2d8b9bc, a=1 (before)
f(): b=0x7ffdd2d8b9bc, b=1 (before)
f(): b=0x7ffdd2d8b9bc, b=50 (after)
main(): &a=0x7ffdd2d8b9bc, a=50 (after)
```

Mixing pointers and references

- Sometimes it is useful to use a reference with a pointer
- Let's assume you get a pointer from somewhere
- And you need to write a lot of code with that pointer
- With a reference you can store the dereferenced pointer and give it a name

```
int * some_pointer = ...;

int & data = *some_pointer;

data = 50;
// instead of *some_pointer = 50;
```

TEMPLE UNIVERSITY

Outline

Scope and Lifetime - Part 2
- Calling functions with pointer arguments
- References
- Calling functions with reference arguments

Pointers and Arrays
- Pointers of array variables

Pointer Arithmetic

Dynamic Memory Allocation

TEMPLE UNIVERSITY

Recap: Addresses of variables on the stack

```
int main() {
    int a = 30;
    int b = 50;
    int c = 40;

    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n", &c);
    return 0;
}
```

Output

```
&a = 0x7ffd47793adc
&b = 0x7ffd47793ad8
&c = 0x7ffd47793ad4
```

- `&a > &b`
- `&b > &c`
- with each variable, their memory addresses grow towards smaller numbers

Array Variables

```c
int main() {
    int a = 30;
    int b = 50;
    float v[3];

    printf("&a = %p\n", &a);
    printf("&b = %p\n\n", &b);

    printf("&v[0] = %p\n", &v[0]);
    printf("&v[1] = %p\n", &v[1]);
    printf("&v[2] = %p\n", &v[2]);
    return 0;
}
```

Array Variables

```
int main() {
    int a = 30;
    int b = 50;
    float v[3];

    printf("&a = %p\n", &a);
    printf("&b = %p\n\n", &b);

    printf("&v[0] = %p\n", &v[0]);
    printf("&v[1] = %p\n", &v[1]);
    printf("&v[2] = %p\n", &v[2]);
    return 0;
}
```

Output

```
&a = 0x7ffd47793adc
&b = 0x7ffd47793ad8

&v[0] = 0x7ffd47793ac0
&v[1] = 0x7ffd47793ac4
&v[2] = 0x7ffd47793ac8
```

- the memory addresses of array elements are increasing
- `&v[0] < &v[1] < &v[2]`
- Δ = `sizeof(float)` = 4
- $\Rightarrow$ consecutive memory

TEMPLE UNIVERSITY

Array Variables

```c
int main() {
    int a = 30;
    int b = 50;
    float v[3];

    printf("&a = %p\n", &a);
    printf("&b = %p\n\n", &b);

    printf("&v[0] = %p\n", &v[0]);
    printf("&v[1] = %p\n", &v[1]);
    printf("&v[2] = %p\n\n", &v[2]);

    printf("&v = %p\n", &v);
    printf(" v = %p\n",  v);
    return 0;
}
```

Output

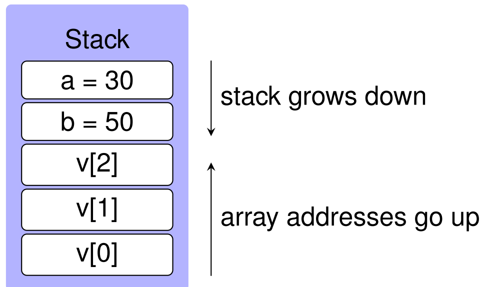
```
&a = 0x7ffd47793adc
&b = 0x7ffd47793ad8

&v[0] = 0x7ffd47793ac0
&v[1] = 0x7ffd47793ac4
&v[2] = 0x7ffd47793ac8

&v = 0x7ffd47793ac0
 v = 0x7ffd47793ac0
```

- `v` and `&v` are a pointer to `v[0]`

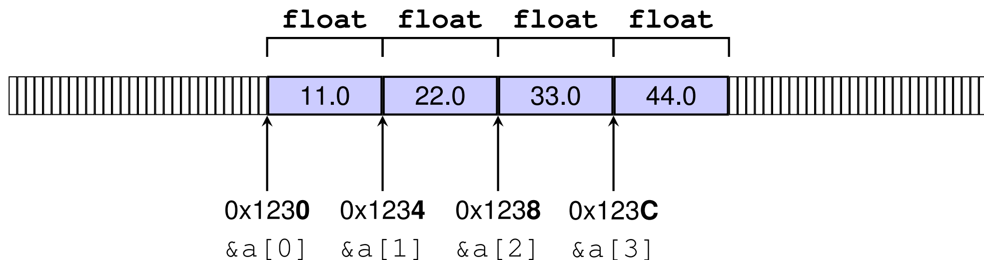
TEMPLE UNIVERSITY

Array Variables

```c
int main() {
    int a = 30;
    int b = 50;
    float v[3];

    printf("&a = %p\n", &a);
    printf("&b = %p\n\n", &b);

    printf("&v[0] = %p\n", &v[0]);
    printf("&v[1] = %p\n", &v[1]);
    printf("&v[2] = %p\n\n", &v[2]);

    printf("&v = %p\n", &v);
    printf(" v = %p\n",  v);
    return 0;
}
```



Pointers of array elements



Computing the address of an array element

$$\underbrace{\text{addr}_i}_{\text{address of element } i} = \underbrace{\text{addr}_0}_{\text{address of first element}} + \underbrace{i}_{\textit{offset}} \cdot \underbrace{\Delta\text{element}}_{\text{size of each element}}$$

$\Delta\text{element} = 4$ (e.g., for **`float`**)

Decimal	Hexadecimal
$\text{addr}_0 = 10000$	$\text{addr}_0 = 0x2710$
$\text{addr}_1 = 10004$	$\text{addr}_1 = 0x2714$
$\text{addr}_2 = 10008$	$\text{addr}_2 = 0x2718$
$\text{addr}_3 = 10012$	$\text{addr}_3 = 0x271C$

TEMPLE UNIVERSITY

Addresses of array elements

```
float a[100];
&a[0];  // computes a + 0*sizeof(float)
&a[1];  // computes a + 1*sizeof(float)
&a[2];  // computes a + 2*sizeof(float)
&a[3];  // computes a + 3*sizeof(float)
```

```
// legal address and compiles
&a[100];
&a[1000];

// but crashes if you try to access it
// because outside of array memory
a[100] = 2.0;   // out of range!
a[1000] = 3.0;  // out of range!
```

Summary

- an array variable is just a pointer to the first element of the array
- using indexing you are accessing a memory location starting from the first element plus an offset
- there are no safe guards for accessing beyond array bounds

Outline

Scope and Lifetime - Part 2

- Calling functions with pointer arguments
- References
- Calling functions with reference arguments

Pointers and Arrays

- Pointers of array variables

Pointer Arithmetic

Dynamic Memory Allocation

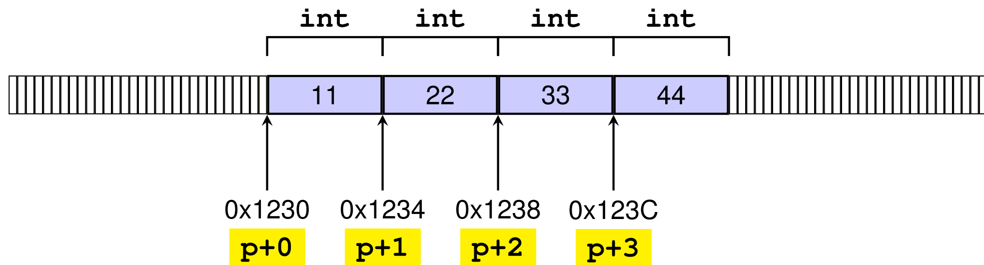
Pointer Arithmetic

```
int a[] { 11, 22, 33, 44};
int * b = &a[0];

b += 1; // + 1*sizeof(int)

// now b points to a[1]

b++; // + 1*sizeof(int)

// now b points to a[2]

b = a + 3; // a + 3*sizeof(int)

// now b points to a[3]
```

- arithmetic operations on pointers have special meaning
- incrementing/decrementing a pointer is done in **multiples of the data type size**
- the compiler does this multiplication for you
- this is the reason why a pointer is dependent on the data type: so the compiler can know the size it has to multiply with

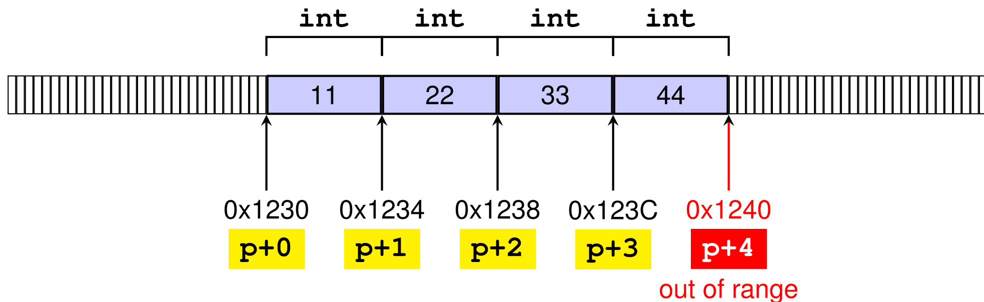
Pointer Arithmetic

```
int a[] {11, 22, 33, 44};
int * p = &a[0];
```



Pointer Arithmetic

```
int a[] {11, 22, 33, 44};
int * p = &a[0];
```



TEMPLE UNIVERSITY

Live Demo: Pointer magic

Example

```
#include <stdio.h>

int main() {
    char buffer[20];
    char text[] = "Hello World!";

    char * a = &text[0];
    char * b = &buffer[0];

    while(*a) {
        *b++ = *a++;
    }

    *b = 0;

    printf("text   = %s\n", text);
    printf("buffer = %s\n", buffer);

    return 0;
}
```

TEMPLE UNIVERSITY

Using array notation with pointers

- It is allowed to use array notation with any pointer
- this follows the same semantics as with arrays: the index is used as the offset which is multiplied with the data type size
- you can use negative indices if you know your pointer is still valid going backwards

Array vs. Pointer notation:

```
int * p = ...;

*p = 10;
// or
p[0] = 10;

*(p+1) = 20;
// or
p[1] = 20;
```

Using negative indices:

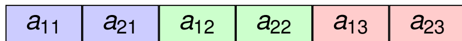
```
int a[4];
int * p = &a[2];

p[-1] = 10; // set a[1]
```

TEMPLE UNIVERSITY

Memory Layouts of Multi-dimensional Arrays

Column-major order

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

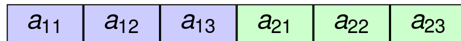


Used by:

- Fortran
- MATLAB
- Julia
- R

Row-major order

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$



Used by:

- C/C++/C#
- Mathematica
- Pascal
- NumPy

Live Demo: Array notation

Example: Array notation

Pointer Notation

```c
int main() {
    int matrix[2][3] { {11, 12, 13},
                       {21, 22, 23} };

    int * p = &matrix[0][0];

    for(int i = 0; i < 6; ++i) {
        printf("%d\n", *p++);
    }
    return 0;
}
```

Array Notation

```c
int main() {
    int matrix[2][3] { {11, 12, 13},
                       {21, 22, 23} };

    int * p = &matrix[0][0];

    for(int i = 0; i < 6; ++i) {
        printf("%d\n", p[i]);
    }
    return 0;
}
```

- prefer array notation because it makes your code more readable

Example: Accessing a 2D array as 1D array

```
#include <stdio.h>

int main() {
    int matrix[2][3] { {11, 12, 13},
                       {21, 22, 23} };

    int * p = &matrix[0][0];

    for(int row = 0; row < 2; ++row) {
        for(int col = 0; col < 3; ++col) {
            int offset = 3*row + col;
            printf("%d\n", p[offset]);
        }
    }
    return 0;
}
```

TEMPLE UNIVERSITY

- Q: We want to write functions which manipulate arrays. If we pass an array to a function, what will happen? Will it be copied?

- Q: We want to write functions which manipulate arrays. If we pass an array to a function, what will happen? Will it be copied?
- Let's find out!

Example: Arrays as function parameters

```
int main() {
    float e1[3];
    float e2[3];
    float e3[3];

    base_vector(e1, 1);
    base_vector(e2, 2);
    base_vector(e3, 3);

    print_vector(e1);
    print_vector(e2);
    print_vector(e3);
    return 0;
}
```

Output

```
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000
```

Example: Arrays as function parameters

```c
int main() {
    float e1[3];
    float e2[3];
    float e3[3];

    base_vector(e1, 1);
    base_vector(e2, 2);
    base_vector(e3, 3);

    print_vector(e1);
    print_vector(e2);
    print_vector(e3);
    return 0;
}
```

```c
void base_vector(float v[3], int idim) {
    for(int i = 0; i < 3; ++i) {
        if(i+1 == idim) {
            v[i] = 1.0;
        } else {
            v[i] = 0.0;
        }
    }
}
```

```c
void print_vector(float v[3]) {
    for(int i = 0; i < 3; ++i) {
        printf("%f ", v[i]);
    }
    printf("\n");
}
```

- Q: We want to write functions which manipulate arrays. If we pass an array to a function, what will happen? Will it be copied?
- A: Arrays are pointers, so only a pointer is copied, not its contents. We can even rewrite the program like this, which is the same:

TEMPLE UNIVERSITY

Example: Arrays as function parameters

```
int main() {
    float e1[3];
    float e2[3];
    float e3[3];

    base_vector(e1, 1);
    base_vector(e2, 2);
    base_vector(e3, 3);

    print_vector(e1);
    print_vector(e2);
    print_vector(e3);
    return 0;
}
```

```
void base_vector( float * v, int idim) {
    for(int i = 0; i < 3; ++i) {
        if(i+1 == idim) {
            v[i] = 1.0;
        } else {
            v[i] = 0.0;
        }
    }
}
```

```
void print_vector( float * v) {
    for(int i = 0; i < 3; ++i) {
        printf("%f ", v[i]);
    }
    printf("\n");
}
```

TEMPLE UNIVERSITY

Example: Arrays as function parameters

- One limitation of our current functions is that they assume 3 dimensions
- Let’s generalize these functions!

```
void base_vector(float * v, int ndim, int idim) {
    for(int i = 0; i < ndim; ++i) {
        if(i+1 == idim) {
            v[i] = 1.0;
        } else {
            v[i] = 0.0;
        }
    }
}
```

```
void print_vector(float * v, int ndim) {
    for(int i = 0; i < ndim; ++i) {
        printf("%f ", v[i]);
    }
}
```

TEMPLE UNIVERSITY

Example: Arrays as function parameters

- Is common in C/C++ code to pass both the size and a pointer
- Our functions are now more resuable

```
int main() {
    float e1[3];
    float e3[5];
    float e7[9];

    base_vector(e1, 3, 1);
    base_vector(e3, 5, 3);
    base_vector(e7, 9, 7);

    print_vector(e1, 3);
    print_vector(e3, 5);
    print_vector(e7, 9);
    return 0;
}
```

Arrays as function parameters: Summary

- Large objects such arrays are passed by their address
- This avoids costly copy operations
- And allows directly modifying existing chunks of data
- the address alone, however, does not give us enough information about the size, so we usually have to pass another parameter if we want to make sure we don't create invalid pointers

Example: Bubble sort

TEMPLE UNIVERSITY

Example: Bubble sort

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

int main() {
    int a[10];

    init_random_list(a, 10);
    print_list(a, 10);

    bubble_sort(a, 10);

    print_list(a, 10);
    return 0;
}
```

```
void swap(int & a, int & b) {
    int tmp = a;
    a = b;
    b = tmp;
}

void bubble_sorts(int * a, int sz) {
    for(int i=0; i< sz; ++i) {
        for(int j=i+1; j < sz; ++j) {
            if(a[j] < a[i]) {
                swap(a[i], a[j]);
            }
        }
    }
}
```

Side Note: Sorting

- Bubble sort is one of the worst sorting algorithms
- Computer scientists have come up with a large selection of different sorting algorithms, optimized for different situations
- Some of them are demonstrated in this nice visualization

Open Questions

Problem 1:

- any data we manipulate lives on the stack
- once we leave a function's stack frame, that data is destroyed
- $\Rightarrow$ any pointer to data inside the function becomes invalid

```
int * f() {
    int a = 5;
    return &a;
} // a becomes invalid

...
int * b = f();
// invalid pointer after f()
// we can not use b anymore
```

Open Questions

Problem 2:

- all our arrays have a fixed size
- that size is determined during **compile time**
- so far we don't know any way to create arrays during **execution time**
  - needed for data from files
  - needed for data from users
  - or any other outside source

```
int a[100];

// what if we need 1000 elements
// while the program runs
// because a user asks for it?

// right now, with what we know,
// we have to change it
int a[1000];
// and recompile our program
```

Outline

Scope and Lifetime - Part 2
- Calling functions with pointer arguments
- References
- Calling functions with reference arguments

Pointers and Arrays
- Pointers of array variables

Pointer Arithmetic

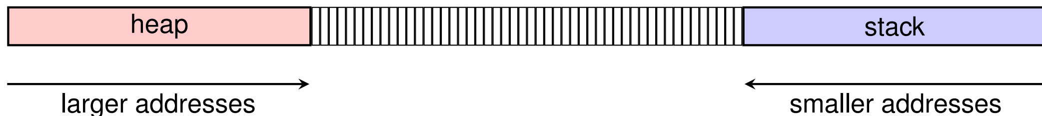
Dynamic Memory Allocation

TEMPLE UNIVERSITY

Memory Space of your program

Stack: grows towards smaller memory addresses

Heap: grows towards larger memory addresses



Stack vs. Heap

Stack

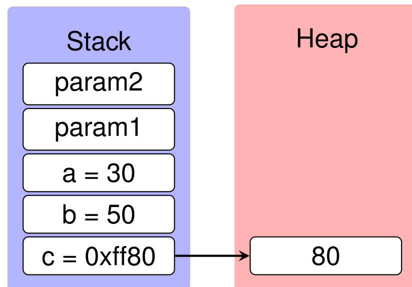
- + fast
- + automatic cleanup
- - not persistent
- - limited storage (limited by OS/Compiler)

Heap

- + persistent
- + “infinite storage” (limited by RAM)
- - slower to acquire
- - manual cleanup

Heap Memory

```
void foo(int param1, int param2)
{
    int a = 30;
    int b = 50;
    int * c = new int;
    *c = 80;
}
```



TEMPLE UNIVERSITY

Single Allocation: C vs. C++

C

```c
// create new int on heap
int * a = (int*)malloc(sizeof(int));

// if a == NULL, we're out of memory
// otherwise, we have an int!

// TODO use our int!

// remove int using address
free(a);

// a is now invalid
// it's a good idea to set to NULL
a = NULL;
```

C++

```cpp
// create new int on heap
int * a = new int;

// if a == NULL, we're out of memory
// otherwise, we have an int!

// TODO use our int!

// remove int using address
delete a;

// a is now invalid
// it's a good idea to set to NULL
a = NULL;
```

```
int * f() {
    int c = 20;
    int * b = new int;
    *b = 50;
    printf("b=%p, *b=%d\n", b, *b);
    return b;
}

int main() {
    int * a = NULL;

    a = f();

    printf("*a = %d\n", *a);

    delete a;
    a = NULL;

    return 0;
}
```

Stack

main()

a = 0x0000

Heap

```
int * f() {
    int c = 20;
    int * b = new int;
    *b = 50;
    printf("b=%p, *b=%d\n", b, *b);
    return b;
}

int main() {
    int * a = NULL;

    a = f();

    printf("*a = %d\n", *a);

    delete a;
    a = NULL;

    return 0;
}
```

Stack

main()

a = 0x0000

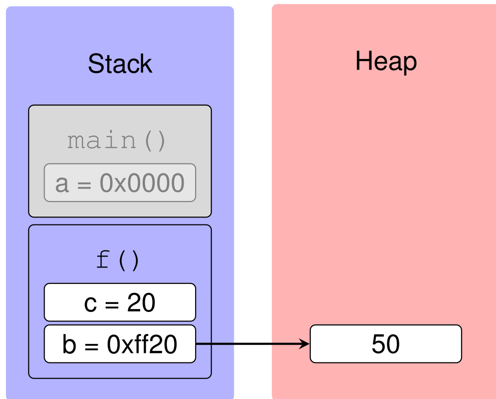
f()

c = 20

Heap

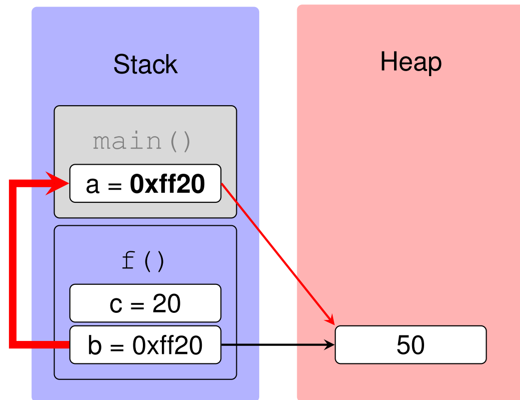
TEMPLE UNIVERSITY

```
int * f() {
    int c = 20;
    int * b = new int;
    *b = 50;
    printf("b=%p, *b=%d\n", b, *b);
    return b;
}

int main() {
    int * a = NULL;

    a = f();

    printf("*a = %d\n", *a);

    delete a;
    a = NULL;

    return 0;
}
```

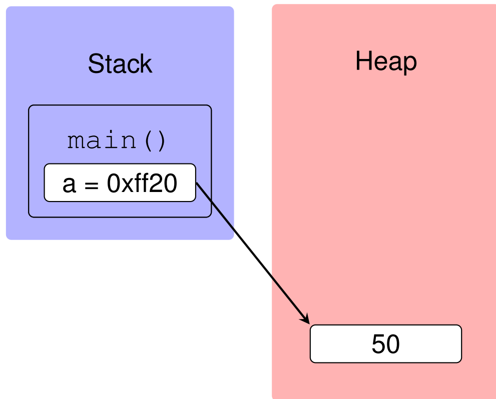


TEMPLE UNIVERSITY

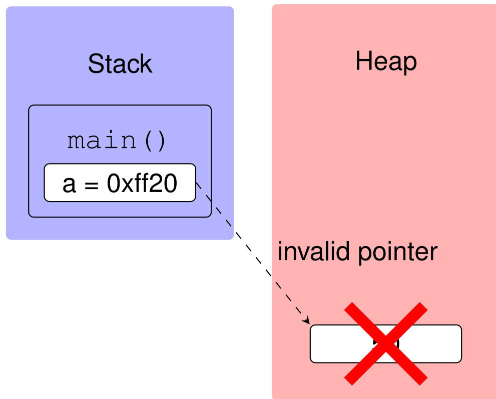
```
int * f() {
    int c = 20;
    int * b = new int;
    *b = 50;
    printf("b=%p, *b=%d\n", b, *b);
    return b;
}

int main() {
    int * a = NULL;

    a = f();

    printf("*a = %d\n", *a);

    delete a;
    a = NULL;

    return 0;
}
```



```
int * f() {
    int c = 20;
    int * b = new int;
    *b = 50;
    printf("b=%p, *b=%d\n", b, *b);
    return b;
}

int main() {
    int * a = NULL;

    a = f();

    printf("*a = %d\n", *a);

    delete a;
    a = NULL;

    return 0;
}
```



```
int * f() {
    int c = 20;
    int * b = new int;
    *b = 50;
    printf("b=%p, *b=%d\n", b, *b);
    return b;
}

int main() {
    int * a = NULL;

    a = f();

    printf("*a = %d\n", *a);

    delete a;
    a = NULL;

    return 0;
}
```



```
int * f() {
    int c = 20;
    int * b = new int;
    *b = 50;
    printf("b=%p, *b=%d\n", b, *b);
    return b;
}

int main() {
    int * a = NULL;

    a = f();

    printf("*a = %d\n", *a);

    delete a;
    a = NULL;

    return 0;
}
```

Stack

main()

a = 0x0000

Heap

TEMPLE UNIVERSITY

Array Allocation: C vs. C++

C

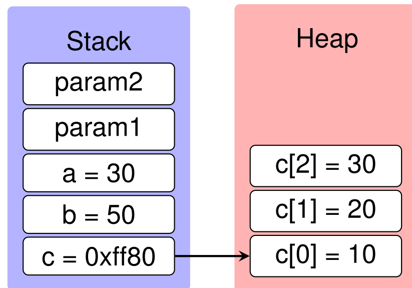
```c
// create 5 new ints on heap
// and return address of first
int* a=(int*)malloc( 5*sizeof(int) );

// if a == NULL, we're out of memory
// otherwise, we have 5 ints!

// TODO use our ints!

// remove memory using address
free(a);

// a is now invalid
// it's a good idea to set to NULL
a = NULL;
```

C++

```cpp
// create 5 new ints on heap
// and return address of first
int * a = new int[5];

// if a == NULL, we're out of memory
// otherwise, we have 5 ints!

// TODO use our ints!

// remove int using address
delete [] a;

// a is now invalid
// it's a good idea to set to NULL
a = NULL;
```

Heap Memory: Arrays

```
void foo(int param1, int param2)
{
    int a = 30;
    int b = 50;
    int * c = new int[3];
    c[0] = 10;
    c[1] = 20;
    c[1] = 30;
}
```



Working with heap memory

Bad news:

- You are now responsible of keeping track of pointers
- Loosing a pointer creates a **memory leak**
- Once memory is leaked, you can not retrieve it
- Eventually you will run out of memory
- $\Rightarrow$ always keep a copy of a pointer so you can call **`delete`**
- $\Rightarrow$ remove data from heap if it is no longer needed
- $\Rightarrow$ set invalid pointers to `NULL` immediately

Working with heap memory

Good news:

- You already know everything about how to use heap memory
- Everything you learned so far about arrays and pointers on the stack is still true when using the heap memory
- Heap memory grows towards larger addresses and **`new`** returns the memory address of the first element, just like arrays on the stack

TEMPLE UNIVERSITY

Final Example: Dynamic Memory Matrix

Example: Dynamic Memory Matrix

```
#include <stdio.h>

float * identity_matrix(int m, int n);
void print_matrix(float * matrix, int m, int n);

int main() {
  int m;
  int n;

  printf("Enter M:");
  scanf("%d", &m);
  printf("Enter N:");
  scanf("%d", &n);

  float * matrix = identity_matrix(m, n);
  print_matrix(matrix, m, n);

  delete [] matrix;
  matrix = NULL;

  return 0;
}
```

TEMPLE UNIVERSITY

Example: Dynamic Memory Matrix

```
float * identity_matrix(int m, int n) {
    float * matrix = new float[n*m];

    for(int j = 0; j < m; ++j) {
        for(int i = 0; i < n; ++i) {
            int offset = n*j + i;
            if (i == j)
                matrix[offset] = 1.0;
            else
                matrix[offset] = 0.0;
        }
    }
    return matrix;
}
```

```
void print_matrix(float * matrix,
                  int m, int n) {
    for(int j = 0; j < m; ++j) {
        for(int i = 0; i < n; ++i) {
            int offset = n*j + i;
            printf("%f ", matrix[offset]);
        }
        printf("\n");
    }
}
```

TEMPLE UNIVERSITY