VG101: Introduction to Computer and Programming

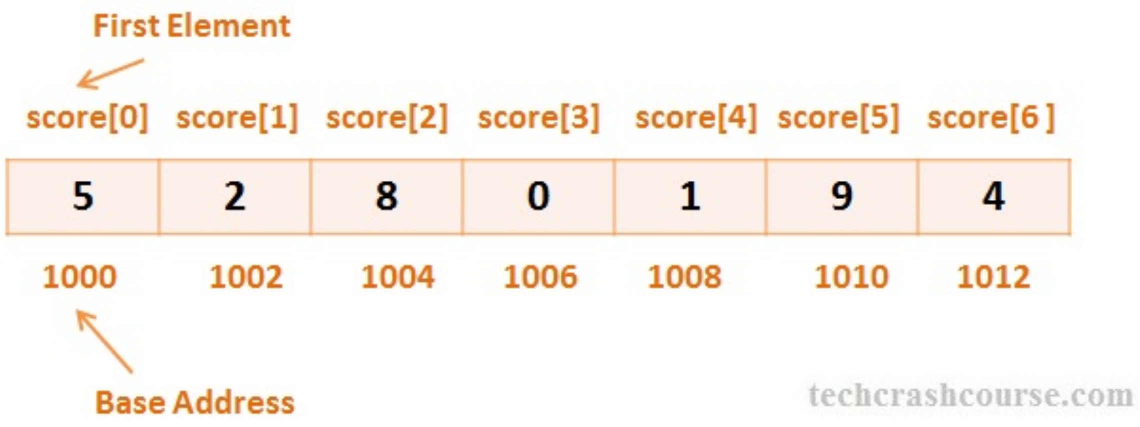
Week8 Checklist

Some further note about HW

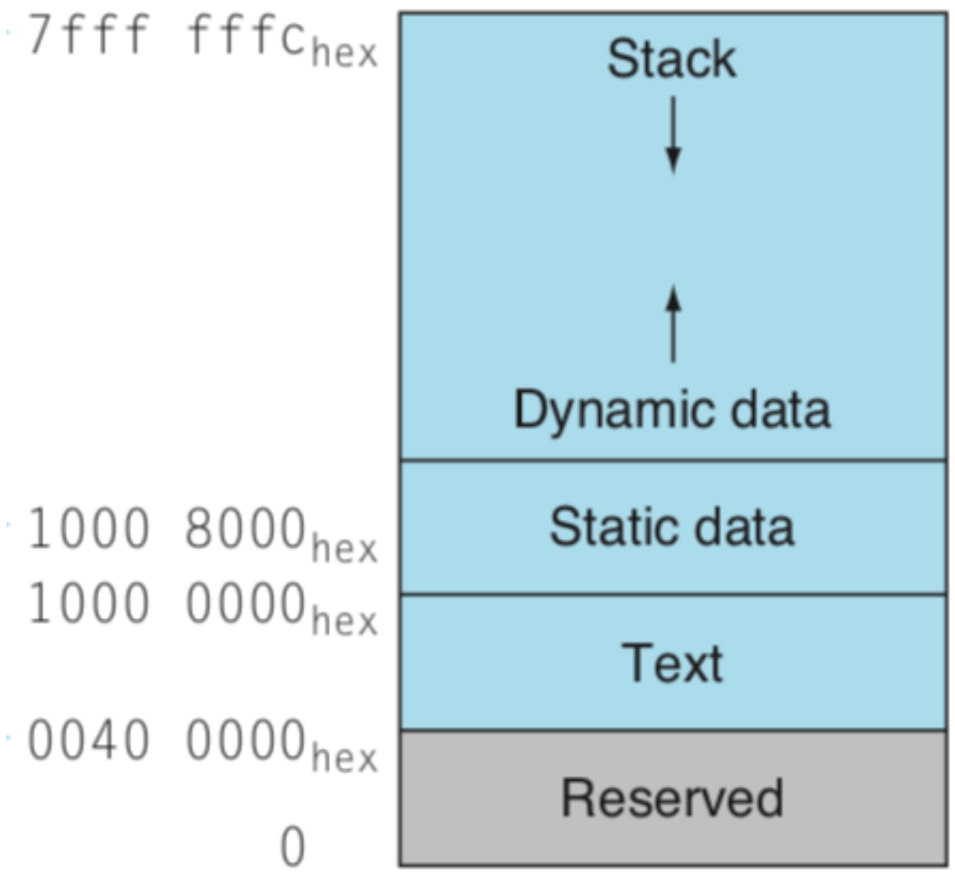
- `double` doesn't store value precisely (float-point value)
- do test your code

Memory

- Basic memory model
- Address



- Aside: Complete memory model



- Reference for further reading
    - https://en.wikipedia.org/wiki/Data_segment

Scope

- Start from declaration, end with the scope (block)
- Local variable
- Global variable
    - declare outside the `main` function
    - global variables are evil
    - Reference for further reading
        - https://stackoverflow.com/questions/484635/are-global-variables-bad
        - https://www.learncpp.com/cpp-tutorial/4-2a-why-global-variables-are-evil/
- Static local variable

```
int func()
{
    static int x = 3;
    x++;
    return x;
}
int main()
```

```c
{
    int y;
    y = func();
    printf("%d\n", y);
    y = func();
    printf("%d\n", y);
    return 0;
}
```

Arrays

- Similar to matrix in MATLAB.
- An **ordered** collection of data values of **the same type**.
    - Why requires same type?

Declaration

- Declare an array: type, name, size.

```c
// Type Name[Size]
int score[5];
char str[30];
double num[20];
```

- Why size is a constant number? Why can't we use a variable?
    - Macro define: `#define MAX_SIZE 100`
- Array in memory space (different type require different size of memory)
- Ways of initialization

```c
int zeros[20] = {0}; // All elements initialized as 0
int notAllZero[20] = {1}; // Only the first one will be 1, other will be 0
int numbers[5] = {1, 2, 3, 4, 5};
int numbers[] = {1, 2, 3, 4, 5}; // Equivalent to previous
char str[4] = {'a', 'b', 'c', '\0'}; // Define a string, '\0' is ASCII 0
int numbers[5];
for (int i=0; i<5; i++)
    numbers[i] = i+1;
```

- Do remember to initial your array (maybe set to 0)!
- Wrong ways of initialization

```c
int numbers_1[5] = {1, 2, 3, 4, 5, 6};// Error: length exceeds
int numbers_2[5];
numbers_2[5] = {1, 2, 3, 4, 5}; // Error: {} only valid when
initialization
numbers_2 = {1, 2, 3, 4, 5};    // Also Error
```

- memory set: `void * memset ( void * ptr, int value, size_t num )`
    - `#include <string.h>`

```
/* memset example */
#include <stdio.h>
#include <string.h>

int main ()
{
  char str[] = "almost every programmer should know memset!";
  memset (str,'-',6);
  puts (str);
  return 0;
}
// output:
// ------ every programmer should know memset!
```

A common usage:

```
char str[20];
memset(str, 0, 20*sizeof(char));
```

Array Element Accessing

- Index starts with 0. (why?)
- Use square bracket `[]`.
- Can use integer variable/expression to as array index

```
int numbers[5];
for (int i=0; i<5; i++)
    numbers[i] = i;     // Note: this is the most common way of iteration
```

- Be careful: array out of bounds! Invalid index can make something **dark magic** happen.

```
int numbers[5];
for (int i=0; i<5; i++)
    numbers[i] = i;
for (int i=0; i<6; i++)
    printf("%d ", number[i]);    // Will not show ERROR!
                                 // C will not check this.
                                 // Will print out data in your memory.
                                 // Or even worse, revise data in your
memory.
                                 // Will cause bugs very hard to find.
```

- How could we visit data outside array? Access through address.
- Question: what will the following code print?

```c
int numbers[5] = {0, 1, 2, 3, 4};
printf("%d\n", numbers);
printf("%p\n", numbers);
```

Array in Function

- Pass array as an argument
- Array pass by address
- Array in function declaration

```c
void ClearIntegerArray(int array[], int length) // In declaration, use array[]
{                                               // Size need to be passed seperately
    for (int i=0; i<length; i++)
        array[i] = 0;
    return;
}

void PrintIntegerArray(int array[], int length)
{
    for (int i=0; i<length; i++)
        printf("%d ", array[i]);
    printf("\n");
    return;
}

void doSomethingMeaningless(int x)
{
    x++;
}

int main()
{
    int x = 5;
    doSomethingMeaningless(x);
    printf("%d\n", x);          // Still 5, because of scope
    int array[5] = {1, 2, 3, 4, 5};
    PrintIntegerArray(array, 5);    // When passing array as argument, no [] needed.
                                    // 1 2 3 4 5
    ClearIntegerArray(array, 5);
    PrintIntegerArray(array, 5);    // 0 0 0 0 0
                                    // Now it is different!
    // Array name itself is an address, address will be passed
    // Pass by address => same array shared!
    return 0;
}
```

Two-dimensional Array

- An array whose elements are arrays.
- Use `A[i][j]` to access elements.
- Row first, column second.
- Stored as one dimensional array in memory.
- Address issue is more complicated.
- Higher-dimensioanl array has same property.

C-style String

- An character array end with `'\0'` (0 in ASCII)
- Always remember to keep a place for `\0` (it is also a char)

```
char str[20] = "Hello World!";  // add a '\0' automatically
                                // so how many chars in this string?
char str2[20];
str2[0] = 'H';
str2[1] = 'E';
str2[2] = 'Y';
str2[3] = '\0';

char str3[20] = {0};
str2[0] = 'H';
str2[1] = 'E';
str2[2] = 'Y';
```

- Concequence if `\0` is missing

Pointer

Previously, we access a piece of memory by the variable name, now we provide another way to access it!

The essence of the pointer

- Pointer is a data type, which store the memory address of a variable
- What is the "address" in the memory?

```c
#include <stdio.h>
int main()
{
    int x = 0;
    int* px = &x;
    printf("%p\n", px);          // 0x7ffeedc037f8
                                 // vary in different computer different
time
    return 0;
}
```

- `*` (dereference) and `&` (reference)
    - `&` requires a lvalue (why?)
    - Aside: lvalue and rvalue
        - lvalue (left value): can be on the left of the assignment expression; keep a memory; a dereferenced pointer is also an lvalue
        - rvalue (right value): expression, constant, etc.
- Pointer declaration

```c
int x, y;       // int variables: x, y
int * px;       // pointer to int: px
                // You can view "int*" as a data type
                // Actually the space near "*" doesn't matter
                // int* px; int *px are all valid
                // Choose the most clear format
int * a, b;
// In this case, a will be a pointer to int, and b will be an int
// DO NOT write code like this in reality
```

- Pointer Assignment

```c
int x;
int* px;
px = &x;                    // Assign the address of x to the px
                            // *px and x are now somehow equivalent
x = 3;
printf("x = %d\n", x);  // x = 3
*px = 4;
printf("x = %d\n", x);  // x = 4
```

- Why pointer require a type?
- `NULL` pointer
    - `NULL == 0`
    - indicate a safe memory
    - cannot be dereference (check before deference if this pointer may be a `NULL`)

- Something tricky: `void*`
    - `void*` cannot be dereference. Why?

```
int x = 3;
int* px = &x;
void* pv = (void*) px;
printf("%d\n", *pv);            // Error!
printf("%d\n", *(int*) pv);     // OK
```

Review the Array by Pointer

- Array name is a pointer!

```
int array[5] = {0, 1, 2, 3, 4};
int* pint_1 = array;
int* pint_2 = &(array[0]);
// pint_1 == pint_2, because the array name is the address of the first
byte of the first element of the array
printf("%p\n", array);          // 0x7ffee0b4a7e0
printf("%p\n", array + 1);      // 0x7ffee0b4a7e4
```

- `[]` is essentially a dereference operator!

```
if (array[3] == *(array + 3))
    printf("There are same!\n");
```

```
- Something evil but legal: `printf("%d", 3[array]);`
- Why index start with 0?
- Why we need to pass the length separately?
- Now you know why `scanf` requires a `&` for `char` but not for an `char`
array: because it requires a pointer (an address to put the scanfed value)
```

- In function call `void PrintIntegerArray(int array[], int length)` is equivalent to `void PrintIntegerArray(int* array, int length)`
- A two dimension array is essentially an array of pointer:

```
int A[5][10];       // A is an int**
                    // Access A[i][j] is a double derefence
```

- The type of a constant C-string `"Hello World!"` is essentially a `const char*`
- Discussion: pass by address vs. pass by value
- Why function has only one return value? What to do if we want more information sending back?

```c
void func(int input_array[], int output_array[]);

int main()
{
    int array1[5] = {0, 1, 2, 3};
    int array2[3];
    func(array1, array2);
    return 0;
}
```

Some Reference for `C`

- https://en.cppreference.com
- `man xxx` for C function