C Programming Language

A Brief Introduction

Dr. Adil ALPKOÇAK

2015

Introduction

- The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s
- Influenced by
  - ALGOL 60 (1960),
  - CPL (Cambridge, 1963),
  - BCPL (Martin Richard, 1967),
  - B (Ken Thompson, 1970)
- Traditionally used for systems programming, though this may be changing in favor of C++
- Traditional C:
  - *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, 2nd Edition, Prentice Hall
  - Referred to as *K&R*

Standard C

- Standardized in 1989 by ANSI (American National Standards Institute) known as ANSI C
- International standard (ISO) in 1990 which was adopted by ANSI and is known as *C89*
- As part of the normal evolution process the standard was updated in 1995 (*C95*) and 1999 (*C99*)
- C++ and C
  - C++ extends C to include support for Object Oriented Programming and other features that facilitate large software development projects
  - C is not strictly a subset of C++, but it is possible to write "*Clean C*" that conforms to both the C++ and C standards.

Why learn C (after Java)?

- Both high-level and low-level language
- Better control of low-level mechanisms
- Performance better than Java (Unix, NT !)
- Java hides many details needed for writing OS code

  But,….
- Memory management responsibility
- Explicit initialization and error detection
- More room for mistakes

C vs. Java

Java	**C**
object-oriented	function-oriented
strongly-typed	can be overridden
polymorphism (+, ==)	very limited (integer/float)
classes for name space	(mostly) single name space, file-oriented
macros are external, rarely used	macros common (preprocessor)
layered I/O model	byte-stream I/O

C vs. Java

Java	**C**
automatic memory management	function calls (C++ has some support)
no pointers	pointers (memory addresses) common
by-reference, by-value	by-value parameters
exceptions, exception handling	if (f() < 0) {error} OS signals
concurrency (threads)	library functions

C vs. Java

Java	**C**
length of array	on your own
string as type	just bytes (char []), with 0 end
dozens of common libraries	OS-defined

C vs. Java

- Java program
  - collection of classes
  - class containing main method is starting class
  - running `java` *`StartClass`* invokes *`StartClass`*`.main` method
  - JVM loads other classes as required

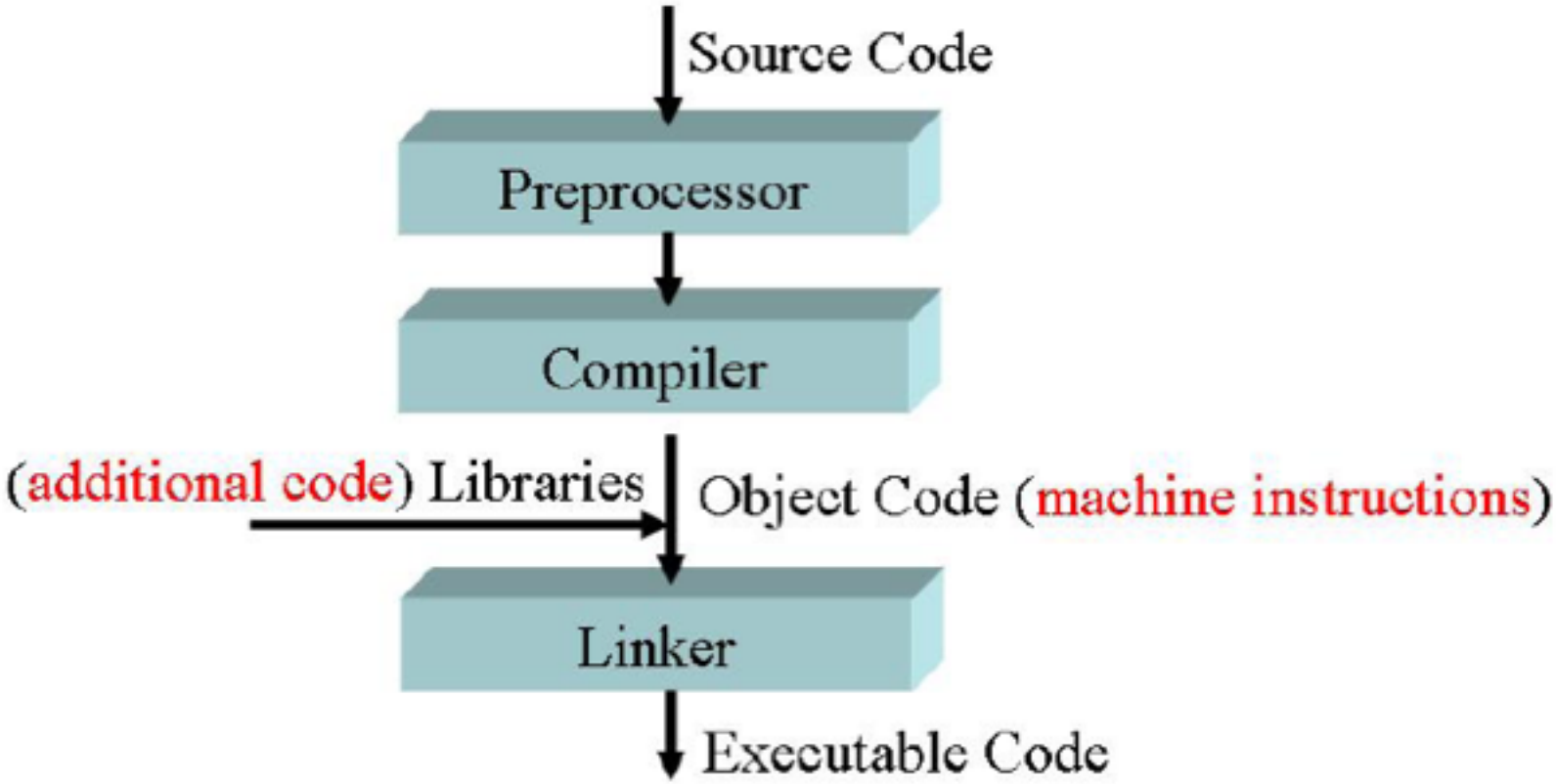
A Simple C Program

```
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   printf("Hello World\n");
   exit(0);
}
```

- *Create* example file: `try.c`
- *Compile* using gcc:
  `gcc -o try try.c`
- The standard C library *libc* is included automatically
- *Execute* program
  `./try`
- Note, I always specify an absolute path
- Normal termination:
  `void exit(int status);`
  - calls functions registered with `atexit()`
  - flush output streams
  - close all open streams
  - return status value and control to host environment

The C Compilation Model



The Preprocessor

- The C preprocessor permits you to define simple macros that are evaluated and expanded prior to compilation.

- Commands begin with a '#'. Abbreviated list:
  - `#define` : defines a macro
  - `#undef` : removes a macro definition
  - `#include` : insert text from file
  - `#if` : conditional based on value of expression
  - `#ifdef` : conditional based on whether macro defined
  - `#ifndef` : conditional based on whether macro is not defined
  - `#else` : alternative
  - `#elif` : conditional alternative
  - `defined()` : preprocessor function: 1 if name defined, else 0

```
#if defined(__NetBSD__)
```

Another Example C Program

/usr/include/stdio.h

```
/* comments */
#ifndef _STDIO_H
#define _STDIO_H

... definitions and protoypes

#endif
```

/usr/include/stdlib.h

```
/* prevents including file
 * contents multiple
 * times */
#ifndef _STDLIB_H
#define _STDLIB_H

... definitions and protoypes

#endif
```

`#include` directs the preprocessor to "include" the contents of the file at this point in the source file.

`#define` directs preprocessor to define macros.

example.c

```
/* this is a C-style comment
 * You generally want to palce
 * all file includes at start of file
 * */
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char **argv)
{
  // this is a C++-style comment
  // printf prototype in stdio.h
  printf("Hello, Prog name = %s\n",
              argv[0]);
  exit(0);
}
```

C Standard Header Files you may want to use

- Standard Headers you should know about:
  - `stdio.h` – file and console (also a file) IO: *p*`error`, `printf`, `open`, `close`, `read`, `write`, `scanf`, etc.
  - `stdlib.h` - common utility functions: `malloc`, `calloc`, `strtol`, `atoi`, etc
  - `string.h` - string and byte manipulation: `strlen`, `strcpy`, `strcat`, `memcpy`, `memset`, etc.
  - `ctype.h` – character types: `isalnum`, `isprint`, `isupport`, `tolower`, etc.
  - `errno.h` – defines `errno` used for reporting system errors
  - `math.h` – math functions: `ceil`, `exp`, `floor`, `sqrt`, etc.
  - `signal.h` – signal handling facility: `raise`, `signal`, etc
  - `stdint.h` – standard integer: `intN_t`, `uintN_t`, etc
  - `time.h` – time related facility: `asctime`, `clock`, `time_t`, etc.

Preprocessor: Macros

- Using macros as functions, exercise caution:
  - flawed example: `#define mymult(a,b) a*b`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = i - 1 * j + 5;`
  - better: `#define mymult(a,b) (a)*(b)`
    - Source: `k = mymult(i-1, j+5);`
    - Post preprocessing: `k = (i - 1)*(j + 5);`
- Be careful of *side effects*, for example what if we did the following
  - Macro: `#define mysq(a) (a)*(a)`
  - flawed usage:
    - Source: `k = mysq(i++)`
    - Post preprocessing: `k = (i++)*(i++)`
- Alternative is to use inline'ed functions
  - `inline int mysq(int a) {return a*a};`
  - `mysq(i++)` works as expected in this case.

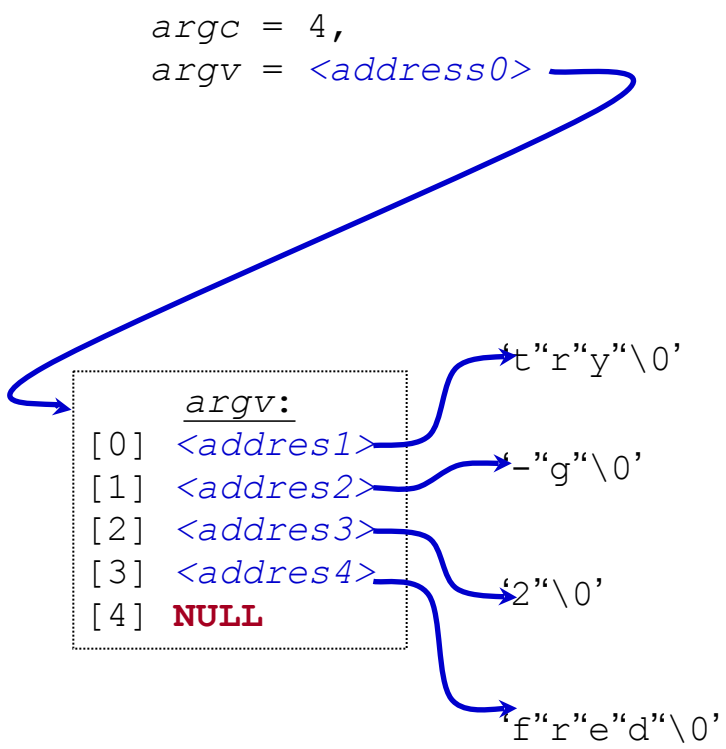
Preprocessor: Conditional Compilation

- Its generally better to use inline'ed functions
- Typically you will use the preprocessor to define constants, perform conditional code inclusion, include header files or to create shortcuts
- `#define DEFAULT_SAMPLES  100`
- `#ifdef __linux`
  ```
  static inline int64_t
    gettime(void) {...}
  ```
- `#elif defined(sun)`
  ```
  static inline int64_t
    gettime(void) {return (int64_t)gethrtime()}
  ```
- `#else`
  ```
  static inline int64_t
    gettime(void) {... gettimeofday()...}
  ```
- `#endif`

Passing Command Line Arguments

- When you execute a program you can include arguments on the command line.
- The run time environment will create an argument vector.
  - `argv` is the argument vector
  - `argc` is the number of arguments
- Argument vector is an array of pointers to strings.
- a *string* is an array of characters terminated by a binary 0 (NULL or '\0').
- *argv[0]* is always the program name, so *argc* is at least 1.

```
./try -g 2 fred
```



Another Simple C Program

```
int main (int argc, char **argv) {
   int i;
   printf("There are %d arguments\n", argc);
   for (i = 0; i < argc; i++)
      printf("Arg %d = %s\n", i, argv[i]);

   return 0;
}
```

- Notice that the syntax is similar to Java
- What's new in the above simple program?
  - of course you will have to learn the new interfaces and utility functions defined by the C standard and UNIX
  - Pointers will give you the most trouble

Basic Types and Operators

- Basic data types
  - Types: *char, int, float and double*
  - Qualifiers: *short, long, unsigned, signed, const*
- Constant: 0x1234, 12, “Some string”
- Enumeration:
  - Names in different enumerations must be distinct
  - ```
    enum WeekDay_t {Mon, Tue, Wed, Thur, Fri};
    enum WeekendDay_t {Sat = 0, Sun = 4};
    ```
- Arithmetic: +, -, *, /, %
  - prefix ++i or --i ; increment/decrement before value is used
  - postfix i++, i--; increment/decrement after value is used
- Relational and logical: <, >, <=, >=, ==, !=, &&, ll
- Bitwise: &, l, ^ (xor), <<, >>, ~(ones complement)
    ```

C keywords

Keywords			
`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Conditional Statements (if/else)

```
if (a < 10)
  printf("a is less than 10\n");
else if (a == 10)
  printf("a is 10\n");
else
  printf("a is greater than 10\n");
```

- If you have compound statements then use brackets (blocks)
  - ```
    if (a < 4 && b > 10) {
      c = a * b; b = 0;
      printf("a = %d, a's address = 0x%08x\n", a, (uint32_t)&a);
    } else {
      c = a + b; b = a;
    }
    ```
- These two statements are equivalent:
  - ```
    if (a) x = 3; else if (b) x = 2; else x = 0;
    ```
  - ```
    if (a) x = 3; else {if (b) x = 2; else x = 0;}
    ```
- Is this correct?
  - ```
    if (a) x = 3; else if (b) x = 2;
    else (z) x = 0; else x = -2;
    ```

Conditional Statements (switch)

```
int c = 10;
switch (c) {
  case 0:
    printf("c is 0\n");
    break;
  ...
  default:
    printf("Unknown value of c\n");
    break;
}
```

- What if we leave the break statement out?
- Do we need the final break statement on the default case?

Functions

- Always use function prototypes

```
int myfunc (char *, int, struct MyStruct *);
int myfunc_noargs (void);
void myfunc_noreturn (int i);
```

- C and C++ are *call by value*, copy of parameter passed to function
  - C++ permits you to specify pass by reference
  - if you want to alter the parameter then pass a pointer to it (or use references in C++)
- If performance is an issue then use inline functions, generally better and safer than using a macro. Common convention
  - define prototype and function in header or *name*.i file
  - `static inline int myinfunc (int i, int j);`
  - `static inline int myinfunc (int i, int j) { ... }`

Structs and Unions

- structures
  - `struct MyPoint {int x, int y};`
  - `typedef struct MyPoint MyPoint_t;`
  - `MyPoint_t point, *ptr;`
  - `point.x = 0;point.y = 10;`
  - `ptr = &point; ptr->x = 12; ptr->y = 40;`
- unions
  - `union MyUnion {int x; MyPoint_t pt; struct {int 3; char c[4]} S;};`
  - `union MyUnion x;`
  - Can only use one of the elements. Memory will be allocated for the largest element

Structures

- Equivalent of Java's classes with only data (no methods)

```
#include <stdio.h>

struct birthday{
    int month;
    int day;
    int year;
  };

main() {
  struct birthday mybday;  /* - no 'new' needed ! */
            /* then, it's just like Java ! */
  mybday.day=1; mybday.month=1; mybday.year=1977;
  printf("I was born on %d/%d/%d", birth.day,
                          birth.month, birth.year);
}
```

More on Structures

```
struct person{
    char name[41];
    int age;
    float height;
    struct {          /* embedded structure */
      int month;
      int day;
      int year;
    } birth;
 };

struct person me;

me.birth.year=1977;……

struct person class[60];
   /* array of info about everyone in class */

class[0].name="Gun"; class[0].birth.year=1971;……
```

Passing/Returning a structure

```
      /* pass struct by value */
void display_year_1(struct birthday mybday) {
  printf("I was born in %d\n", mybday.year);
}        /* - inefficient: why ? */
. . . .
      /* pass struct by reference */
void display_year_2(struct birthday *pmybday) {
  printf("I was born in %d\n", pmybday->year);
   /* warning ! '->', not '.', after a struct pointer*/
}
. . . .
      /* return struct by value */
struct birthday get_bday(void){
  struct birthday newbday;
  newbday.year=1971;  /* '.' after a struct */
  return newbday;
}        /* - also inefficient: why ? */
```

enum - enumerated data types

```
#include <stdio.h>
enum month{
    JANUARY,        /* like #define JANUARY 0 */
    FEBRUARY,       /* like #define FEBRUARY 1 */
    MARCH           /* … */
};

/* JANUARY is the same as month.JANUARY */

/* alternatively, …. */

enum month{
    JANUARY=1,      /* like #define JANUARY 1 */
    FEBRUARY,       /* like #define FEBRUARY 2 */
    MARCH           /* … */
};
```

Expressions and Evaluation

Expressions combine Values using Operators, according to precedence.

```
1 + 2 * 2      → 1 + 4      → 5
(1 + 2) * 2    → 3 * 2      → 6
```

Symbols are evaluated to their Values before being combined.

```
int x=1;
int y=2;
x + y * y      → x + 2 * 2    → x + 4    → 1 + 4    → 5
```

Comparison operators are used to compare values.
In C, 0 means “false”, and *any other value* means “true”.

```
int x=4;
(x < 5)              → (4 < 5)            → <true>
(x < 4)              → (4 < 4)            → 0
((x < 5) || (x < 4))   → (<true> || (x < 4))  → <true>
```

Not evaluated because first clause was true

Comparison and Mathematical Operators

== equal to
< less than
<= less than or equal
> greater than
>= greater than or equal
!= not equal
&& logical and
ll logical or
! logical not

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit. For oft-confused cases, the compiler will give you a warning "Suggest parens around …" – do it!

+ plus
- minus
* mult
/ divide
% modulo

& bitwise and
l bitwise or
^ bitwise xor
~ bitwise not
<< shift left
>> shift right

Beware division:
- If second argument is integer, the result will be integer (rounded):
  5 / 10 → 0 *whereas* 5 / 10.0 → 0.5
- Division by 0 will cause a FPE

Don't confuse & and &&..
1 & 2 → 0 *whereas* 1 && 2 → <true>

Assignment Operators

```
x = y   assign y to x
x++     post-increment x
++x     pre-increment x
x--     post-decrement x
--x     pre-decrement x
```

```
x += y  assign (x+y) to x
x -= y  assign (x-y) to x
x *= y  assign (x*y) to x
x /= y  assign (x/y) to x
x %= y  assign (x%y) to x
```

Note the difference between ++x and x++:

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don't confuse = and ==! The compiler will warn "suggest parens".

```
int x=5;
if (x==6)   /* false */
{
  /* ... */
}
/* x is still 5 */
```

```
int x=5;
if (x=6)   /* always true */
{
  /* x is now 6 */
}
/* ... */
```

recommendation

Operator Precedence (from “C a Reference Manual”, 5th Edition)

Tokens	Operator	Class	Precedence	Associates
names,	simple tokens	primary	16	n/a
a[k]	subscripting	postfix		left-to-right
f(...)	function call	postfix		left-to-right
.	direct selection	postfix		left-to-right
->	indirect selection	postfix		left to right
++ --	increment, decrement	**postfix**		left-to-right
(*type*){*init*}	compound literal	postfix		left-to-right
++ --	increment, decrement	**prefix**	15	right-to-left
sizeof	size	unary		right-to-left
~	bitwise not	unary		right-to-left
!	logical not	unary		right-to-left
- +	negation, plus	unary		right-to-left
&	address of	unary		right-to-left
*	indirection	unary		right-to-left

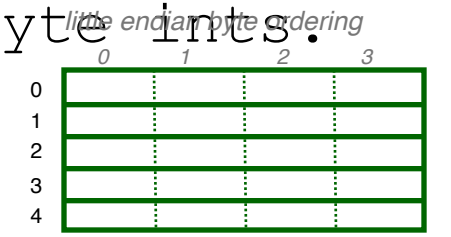
Tokens	Operator	Class	Precedence	Associates
(*type*)	casts	unary	14	right-to-left
* / %	multiplicative	binary	13	left-to-right
+ -	additive	binary	12	left-to-right
<< >>	left, right shift	binary	11	left-to-right
< <= > >=	relational	binary	10	left-to-right
== !=	equality/ineq.	binary	9	left-to-right
&	bitwise and	binary	8	left-to-right
^	bitwise xor	binary	7	left-to-right
|	bitwise or	binary	6	left-to-right
&&	logical and	binary	5	left-to-right
||	logical or	binary	4	left-to-right
?:	conditional	ternary	3	right-to-left
= += -= *= /= %=	assignment	binary	2	right-to-left
,	sequential eval.	binary	1	left-to-right

Arrays and Pointers

- A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

```
int x[5]; // an array of 5 4-byte ints.
```

- All arrays begin with an index of 0



memory layout for array x

- An array identifier is equivalent to a pointer that references the first element of the array
  - `int x[5], *ptr;`
    `ptr = &x[0]` is equivalent to `ptr = x;`
- Pointer arithmetic and arrays:
  - `int x[5];`
    `x[2]` is the same as `*(x + 2)`, the compiler will assume you mean 2 objects beyond element x.

Pointers

- For any type T, you may form a pointer type to T.
  - Pointers may reference a function or an object.
  - The value of a pointer is the address of the corresponding object or function
  - Examples: `int *i; char *x; int (*myfunc)();`
- Pointer operators: ***** dereferences a pointer, *&* creates a pointer (reference to)
  - ```
    int i = 3; int *j = &i;
    *j = 4; printf("i = %d\n", i); // prints i = 4
    ```
  - ```
    int myfunc (int arg);
    int (*fptr)(int) = myfunc;
    i = fptr(4); // same as calling myfunc(4);
    ```
- Generic pointers:
  - Traditional C used (char *)
  - Standard C uses (void *) – these can not be dereferenced or used in pointer arithmetic. So they help to reduce programming errors
- Null pointers: use ***NULL*** or ***0***. *It is a good idea to always initialize pointers to NULL.*

Pointers in C

```
Step 1:
int main (int argc, argv) {
  int  x = 4;
  int *y = &x;
  int *z[4] = {NULL, NULL, NULL, NULL};
  int  a[4] = {1, 2, 3, 4};
...
```

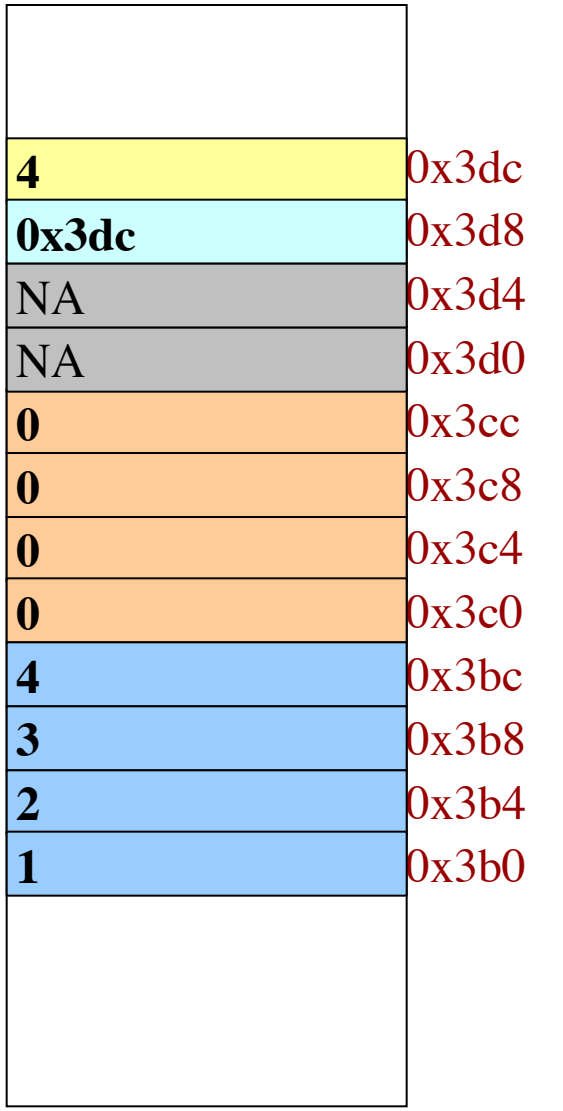
Note: The compiler converts z[1] or *(z+1) to
Value at address (Address of z + sizeof(int));

In C you would write the byte address as:

```
(char *)z + sizeof(int);
```

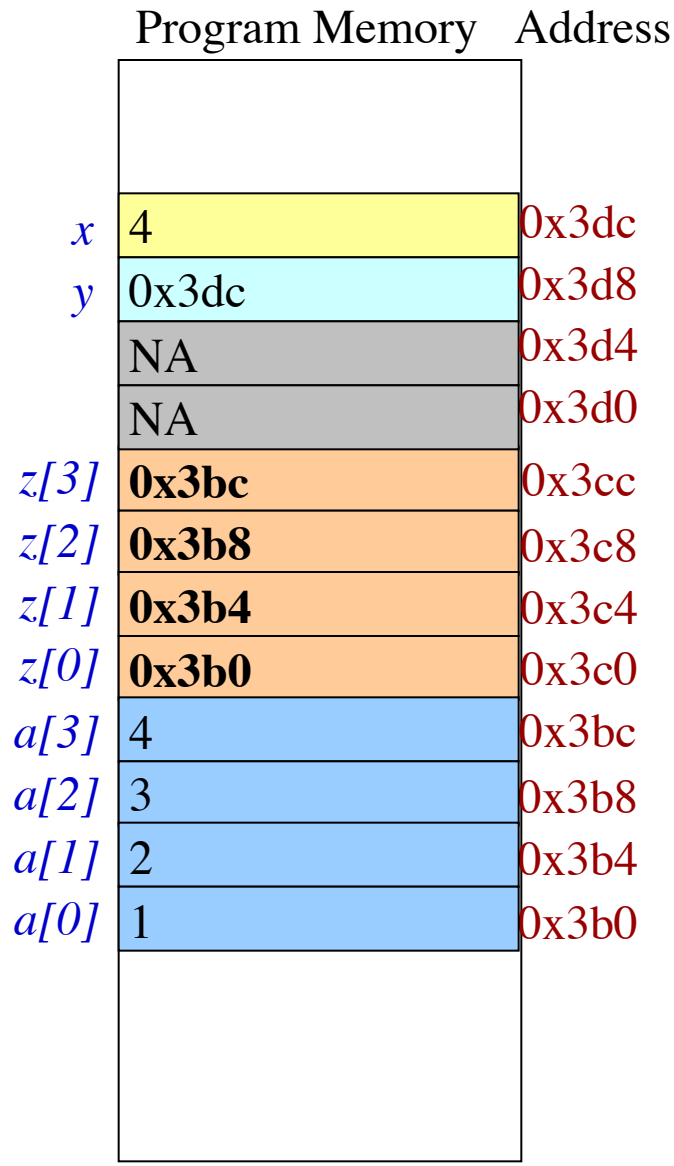
or letting the compiler do the work for you

```
(int *)z + 1;
```



Pointers Continued

```
Step 1:
int main (int argc, argv) {
  int  x = 4;
  int *y = &x;
  int *z[4] = {NULL, NULL, NULL, NULL};
  int  a[4] = {1, 2, 3, 4};
Step 2: Assign addresses to array Z
  z[0] = a;      // same as &a[0];
  z[1] = a + 1;  // same as &a[1];
  z[2] = a + 2;  // same as &a[2];
  z[3] = a + 3;  // same as &a[3];
```



Pointers Continued

```
Step 1:
int main (int argc, argv) {
  int x = 4;
  int *y = &x;
  int *z[4] = {NULL, NULL, NULL, NULL};
  int a[4] = {1, 2, 3, 4};
Step 2:
  z[0] = a;
  z[1] = a + 1;
  z[2] = a + 2;
  z[3] = a + 3;
Step 3: No change in z's values
  z[0] = (int *)((char *)a);
  z[1] = (int *)((char *)a
                 + sizeof(int));
  z[2] = (int *)((char *)a
                 + 2 * sizeof(int));
  z[3] = (int *)((char *)a
                 + 3 * sizeof(int));
```