**Some more features**

**All about constness:**

example: const.c

Rules of programming:

if something should be a constant, make it a constant.

- compiler may replace the use of a constant by its value

- makes the program more efficient

Two points to know about constants

1. const should be initialized

2. const cannot be assigned

While passing a pointer to a variable, make the corresponding parameter a pointer to constant if the argument should not be changed.

- interface becomes more robust. Client knows that his variable cannot be changed by the invoked function

- compiler might be able to optimize the code

// making constants :

// 1. using macro

#define MAX 100

// 2. using const qualifier

const int MIN = -100;

The second is preferred.

// These become more interesting when we have pointers

int main()

{

// pointer to a const

#if 0

const int a = 10; const int b = 20;

// p is a pointer to an int, which is a constant

// p can be changed; not \*p

const int \*p = &a;

p = &b;

//\*p = 30; // compile time error

#endif

#if 0

// const pointer

int a = 10; int b = 20;

// p is a pointer to an int, which is a constant

// p can be changed; not \*p

int \* const p = &a;

// p = &b; // compile time error

\*p = 30;

#endif

#if 0

// const pointer to a const int

int a = 10; int b = 20;

// p is a pointer to an int, which is a constant

// p can be changed; not \*p

const int \* const p = &a;

// p = &b; // compile time error

// \*p = 30; // compile time error

#endif

// conversion of a pointer to a variable to a pointer to a const

// perferctly alright; used very heavily

// we can treat a variable as a constant

// we cannot treat a constant as a variable

int a = 10;

// p is a pointer to an int, which is a constant

// p can be changed; not \*p

const int \*p = &a;

}

Observe that conversion of a pointer to variable to a pointer to constant is trivial. This is very much used in parameter passing.

**enum:**

This allows us to enumerate a list of named or manifest constants – all of int type and no two constants can have the same value.

example: enum.c

#include <stdio.h>

// create a number of manifest constants

// a) create a number of named constants

#if 0

const int a = 0;

const int b = 1;

const int c = 2;

const int d = 3;

int main()

{

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

}

#endif

// b : use enum

// enum stands for enumeration

// equivalent to a number of named constants

// cannot be assiged; compile time error

// leftmost one by default has a value 0

// each one to the right unless initialized has the successor value of the left

#if 0

enum {a, b, c, d};

int main()

{

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

//a = 11; // Error

}

#endif

enum {a = 10, b, c = 20, d};

int main()

{

printf("%d %d %d %d\n", a, b, c, d); // 10 11 20 21

//a = 11; // Error

}

**Variable number of arguments:**

There are functions like printf which can take any number of arguments. These functions indicate this ability by using … in their signature – this is called the ellipses symbol. As types are not specified for these varying number of arguments, the type of argument and parameter cannot be matched. There is no way to infer the type by looking at the bit pattern on the stack as the type information is not stored at runtime. So runtime on its own cannot get the value of the arguments stored as bits on the stack.

C provides an interesting way to solve the problem. We can create a pointer like entity to refer to the one after the last named parameter. The opaque type is called va\_list. Then, we can access and get the right value by specifying the expected type to a macro va\_arg. The variable is advanced properly to the next location in the stack.

Please note that if one of the calls to va\_arg goes bad, the rest will all be garbage.

We should have some logic to know when to stop as well as what type of value we would like to extract.

In printf, the format specifiers like %d %lf decide how to interpret the next bit pattern and the end of the string indicates the end of processing.

example: variable\_args.c

// find sum of n integers

// signature:

// int sum(int n, ...);

// The first argument is an integer.

// Followed by any # of arguments of any type

// Compiler cannot check for types

// This is type unsafe.

// Semantics of our function :

// The first argument indicates the number of integers following it.

// How does this work?

// make a variable of type : va\_list

// initialize it to point to the parameter after the last named parameter

// get the value at the location by using va\_arg. Observe the second argument

// is a type.

// This macro gets the value of the given type from that location adn also

// advances the variable of va\_list to the location of the next parameter

// At the end, break the relationship between the variable and the stack frame

// by using the macro va\_end.

// va\_arg should be given the next expected type as argument. If it does not

// match what is pushed in, we may have unexpected results.

// Remember that the type info is not stored at runtime - value of any bit pattern

// depends on how we interpret it.

#include <stdio.h>

#include <stdarg.h>

int sum(int n, ...); // declaration

int main()

{

printf("sum : %d\n", sum(5, 11, 33, 22, 55, 44));

printf("sum : %d\n", sum(3, 200, 300, 100));

}

// definition

int sum(int n, ...)

{

int s = 0;

va\_list va;

va\_start(va, n);

for(int i = 0; i < n; ++i)

{

s += va\_arg(va, int);

}

va\_end(va);

return s;

}

**Command line arguments:**

It is possible to pass arguments to the function main by specifying them along with the program being executed.

These are received as array of strings.

The entry function main of ‘C’ can have two parameters – the second is a pointer to an array of c strings – array has NULL at the end - array is NULL terminated, the first parameter indicates the count of elements in the array. The zeroth element is the command itself.

example: cmdline.c

// main can have any one of the following signatures

// int main()

// or

// int main(int argc, char\* argv[])

// parameter names are not keywords - but they are normally named as shown

// they carry some interesting abstraction

// When we run the program, we can specify a sequence of strings along with the

// image name

// Example:

// ./a.out we love computer science

// argc : count of arguments including the image; in this case 5

// argv : pointer to an array of pointers to char. Each element of argv points

// to a string.

// We use this concept a lot.

// Example:

// cat a.c b.c c.c

// cp x y

// gcc -c x.c y.c

// simple example : echoes the command line arguments

#include <stdio.h>

int main(int argc, char\* argv[])

{

printf("program name : %s\n", argv[0]);

for(int i = 1; i < argc; ++i) // observe i = 1 initially.

{

printf("%s ", argv[i]);

}

printf("\n");

}

**system**:

We may want to execute a program or a command of the underlying operating system. There is a function called system which takes the command as a string and executes it.

This program illustrates a small shell.

We can keep entering commands and this program will execute them until we key in the string over.

example: system.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

int main()

{

char cmd[80];

scanf("%[^\n]s", cmd);getchar();

while(strcmp(cmd, "over") != 0)

{

system(cmd);

scanf("%[^\n]s", cmd);getchar();

}

}

**environment**:

A program does not execute in isolation. It uses certain settings made by the operating system. This set of variables is called environment variable. These are stored as pairs in the form variable=value. This is inherited when a new process is created and can be changed to affect the new process or any process launched by this process. A process cannot change the environment of its parent.

There are two ways of accessing the environment variables.

a) Use OS specific functions.

Unix provides getenv and setenv.

b) use a variable named environ.

Extern char \*\*environ;

We compile a program called hello.c to a loadable image hello.

If we run by typing hello at the prompt in unix, it will not work. When we give a command which is a filename, the shell searches for this name in a sequence of directories. This sequence of directories is associated with an environment variable called PATH. This does not have . - current directory. We can add this by using setenv function. Then we can run the program hello.

Check the program below.

example: env.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// environment:

// OS provides an environment in which the programs work.

// Examples:

// prompt, present working directory, ...

// $ env

// Each entry is a key value pair - both are strings

// We can access or modify(create) these using getenv and setenv functions.

// This feature is OS dependent

// example:

// PATH

// The value is a sequence of directory names separated by : in unix and ; in microsoft windows

// When we execute a program, the image should be in one of the directories.

// The program below shows how to modify the PATH have . (current directory) as one of the components.

// Now we can execute the programs without prefixing ./

// Also note that this change does not affect the program or the shell running this.

int main()

{

char \*path;

char mypath[1000];

path = getenv("PATH");

printf("path : %s\n", path);

system("hello"); // Error

strcpy(mypath, path);

strcat(mypath, ":.");

setenv("PATH", mypath, 1);

system("hello"); // NO Error

printf("end\n");

}

example: hello.c

Compile hello.c : gcc hello.c -o hello

#include <stdio.h>

int main()

{

printf("hello world\n");

return 0;

}

This is way to display all the environment variables using extern variable environ.

example: env1.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// environment:

// OS provides an environment in which the programs work.

// Examples:

// prompt, present working directory, ...

// $ env

// Each entry is a key value pair - both are strings

// We can access or modify(create) these using getenv and setenv functions.

// This feature is OS dependent

// example:

// PATH

// The value is a sequence of directory names separated by : in unix and ; in microsoft windows

// When we execute a program, the image should be in one of the directories.

// The program below shows how to modify the PATH have . (current directory) as one of the components.

// Now we can execute the programs without prefixing ./

// Also note that this change does not affect the program or the shell running this.

// we do not directly play with this variable.

// We prefer to use the interface getenv and setenv.

// Looping is interesting. Write a diagram to check how it works.

extern char\*\* environ;

int main()

{

char \*\*p = environ;

while(\*p != NULL)

{

printf("%s\n", \*p);

p++;

}

}

**porting*:***

example:port.c

/\*

portability:

making the program work on different compilers or OS or hardware with little or no change.

Normally compiled codes are not portable.

In 'C', we talk about source level portability.

How do we achieve this?

\*/

//1. use standard language

// Do not use extensions

// example:

// nested function is not portable; is a feature of gnu C

#if 0

#include <stdio.h>

int main()

{

void foo() { printf("this is a test\n"); }

foo();

}

#endif

// 2. Do not assume about the size of types. Always use sizeof on the type

// int \*p = (int\*)malloc(4); // dangerous; What if the size of int is not 4 in

// some other compiler running on some other architecture

// guaranteed:

// sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)

// 3. do not depend on order of evaluation of operands in an expression

// and order of evaluation of arguments to a function.

// a[i] = i++; // undefined

// is ambiguous; can give different results depending on compilers and compiler options.

// printf("%c %c", getchar(), getchar());

// If you input a b, output could be a b or b a

// 4. signed and bitwise operations

// -25 => 11001 complement and add 1

// 1111 .... 1110 0110

// shift right 2 times;

// should we fill the vacant slots by 1 or 0?

// signed or logical shift OR magnitude or arithmetic shift

// sign preserving or value preserving?

// Avoid bitwise operations on signed

#if 0

#include <stdio.h>

int main()

{

printf("%d\n", -25 >> 2); // ???

}

#endif

// 5. byte ordering or ending(big endian or little endian)

// matters in interprocess communication

#if 0

#include <stdio.h>

union myunion

{

unsigned short int x;

struct {

unsigned char a;

unsigned char b;

} two;

};

int main()

{

union myunion u;

u.x = 0xABCD;

printf("%x %x\n", u.two.a, u.two.b); // cd ab

}

#endif

// 6. OS dependent features

// use conditional inclusion

// or

// separate files

// 7.

// check version of your software

#include <stdio.h>

int main()

{

printf("gcc version: %d.%d.%d\n",\_\_GNUC\_\_,\_\_GNUC\_MINOR\_\_,\_\_GNUC\_PATCHLEVEL\_\_);

return 0;

}