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

double integrate (double (*fptr)(double), double a, double b) {
    double sum = 0.0, x, delta_x = 0.1;
    for (x = a; x <= b; x = x + delta_x)
        sum = sum + (*fptr)(x) * delta_x;
    return sum;
}

void initial (int x, char* y) {
    printf ("fewer than %d chars in %s\n", x, y);
}
void main (void) {

    double (*fptr)();
                          /* want a pointer to a function not */
    int (*funptr)();      /*a function that returns a pointer */

    funptr = initial;

    initial (64, "Help me, please?") ;
    (*funptr) (24, "Why, Oh why?") ;
                                          /* next example*/
    printf ("Integral of sine is %f\n",
              integrate (sin, 0.0, 1.1));

    fptr = cos;
    printf ("Cosine integral is %f\n",
              integrate ((*fptr), 0.0, 1.1));
}
```

Macros With Parameters

A macro with parameters is defined as follows:

```
#define macro_name(parameters) macro_body
```

No blanks within the `macro_name(parameters)`

When the macro name is used in the program text a value for each parameter must be supplied -

since this is a pure text replacement there is no checking of types only that there is the correct number of parameters

When the macro body is expanded each parameter is replaced by its value

For example:

```
#define  cube(x)   (x)*(x)*(x)
```

This macro will cube whatever expression is provided for x, this is why the parenthesis are required

`cube(a+b)` will be expanded to

```
(a+b)*(a+b)*(a+b)
```

Without the parenthesis in the macro body, `cube(a+b)` will be expanded to

a+b*a+b*a+b

This is definitely not what we want
REMEMBER that macros are pure text replacement they are not function calls -

* Lack of care will cause nasty problems

For example, `cube(a++)` expands in the following way:

```
(a++)*(a++)*(a++)
```

This computes the correct value for the cube, but the new value of a will be:

a+3

after this expression and not a+1 as we might expect

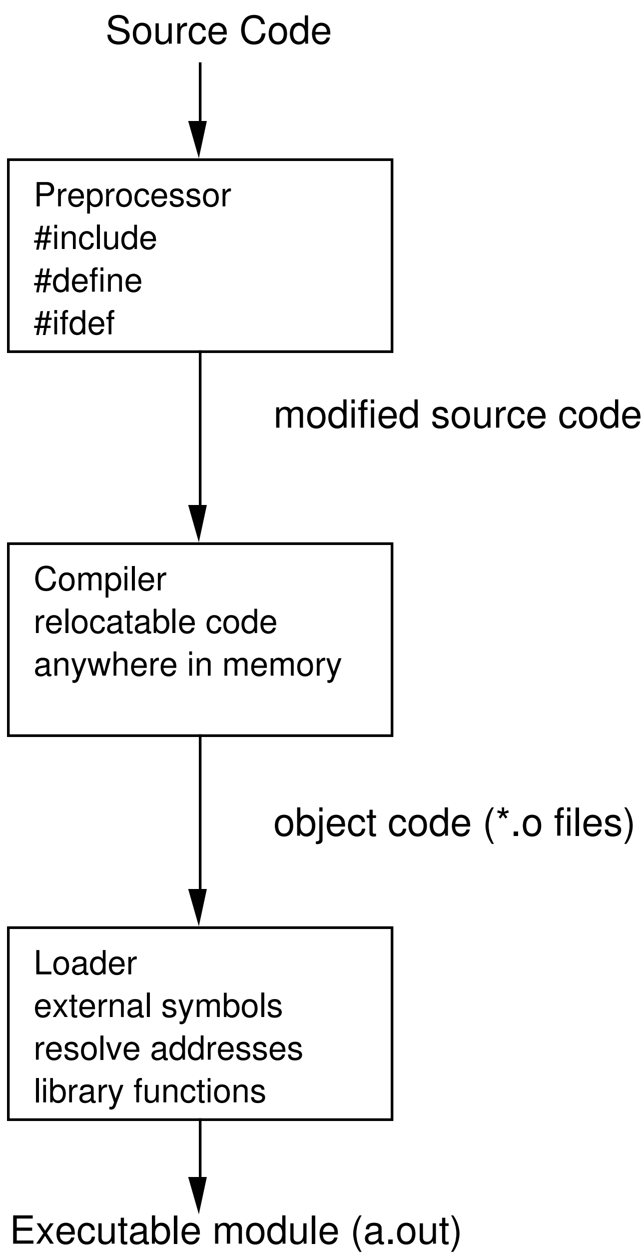
A number of the standard library procedures, like getchar and putchar are really macros and not procedures -

* this saves the overhead of doing a procedure call

A Day in the Life of a C Program

* You compile and execute a C program. Then what??

* Let us run through the entire process and see what happens to the program



* We already know the first thing that happens is the C preprocessor scans the source form of the program and produces a new source file with all the preprocessor directives removed, macros expanded, include files included in the source file, etc.

* This modified version of the source file is then passed to the C compiler to produce the object form of the program, if the name of the source file is xxx.c then the name of the object file will be xxx.o

* The .o file contains relocatable machine language - that is there is enough information in the .o file to place the machine language code anywhere in the address space of the machine

* The next process is loading or link editing the program

LOADING

* The program ld performs the loading of your C program, the input to this program is a collection of .o files and libraries and the output is an executable program

* You usually don't invoke ld directly, the C compiler usually does it for you - the C compiler knows the special libraries and startup code that is required for C programs, you don't need to remember this stuff

* After compilation your C program is on a number of .o files, so all of these files will need to be combined into one file that is the executable version of the program, when the files are combined references between the files must be satisfied

* That is, when a procedure in file1.c calls a procedure in file2.c, the first procedure will eventually need to know that address of the second procedure, when file1.c is compiled this address is unknown so it is left as an external reference

* When the program is loaded, ld finds all the external references, determines where the routines and variables are actually located in memory and then goes back and fills in these addresses

* Loading also includes library files - your program can reference procedures that are in a library, when the program is loaded the procedures in the library must also be loaded, and thus the external references are satisfied

* The -l (ell) flag is used to specify the libraries that are to be loaded with a program, the standard C library is automatically loaded with your program

* For example to load the math library with your program the -lm option must be included when you perform the load operation

* A small assembler program is also loaded with your program, this program sets up the environment for executing your C program, it sets the stack pointer, sets up the values of argc and argv and transfers control to your main() procedure

Properties of the a.out File

* The output from ld is often called an a.out file, since this is the default name of the file

* This is the executable version of the program, this file is basically divided into 3 sections:

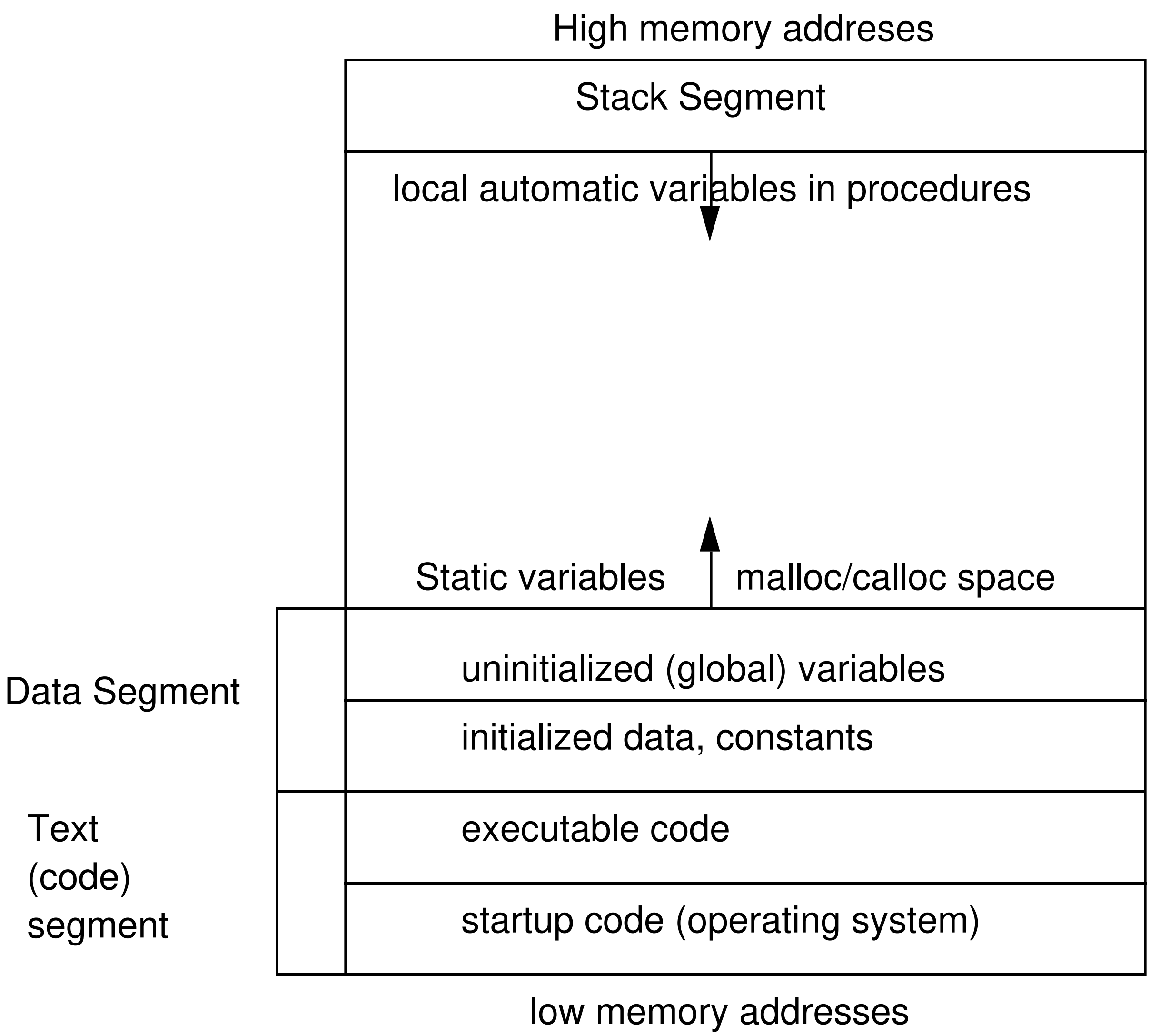
1. text - the machine language or instructions for the program

2. initialized data - the global variables in the program that have been given initial values

3. uninitialized data - the global variables that haven't been assigned initial values

* The a.out file also has a header that specifies how the program is to be executed, see man a.out for a detailed description of this

MEMORY LAYOUT



* When your program is running its memory is divided into three segments, which are:

1. Text segment, 2. Data Segment, 3. Stack Segment

1. text segment - this contains the machine language instructions for your program, depending upon the header information this segment may be read only

2. data segment - this segment contains the global variables for your program, plus any memory that is allocated dynamically while your program is executing

3. stack segment - this segment contains the local variables for the procedures in your program, each time a procedure is called this segment could grow

* Both the data (heap) and stack segment can grow while the program is executing, the stack starts at the highest memory address and the data segment starts just after the text segment, they grow towards each other

Program Execution

* What happens when our program starts executing?

* Control is first transferred to the startup code that is provided by the C compiler and operating system - this code sets up the environment that the program runs in

* In the UNIX environment the startup code first points the stack pointer to the top of the memory space and then prepares to call the main() procedure in your program

* The main procedure should have the following declaration:

```
main (int argc, char* argv[], char* envp[]) {

}
```

* argc is the number of parameters passed to the program, remember that the program name is counted as one of the parameters

* argv is an array of string pointers, there is one entry in this array for each parameter to the program, the parameters to a program are separated by white space, unless they are enclosed in quotes (" or ')

* For example

```
prog -x -f fred george
```

will produce the following:

```
    argc = 5
    argv[0] = "prog"
    argv[1] = "-x"
    argv[2] = "-f"
    argv[3] = "fred"
    argv[4] = "george"
```

* envp is an array of pointers to strings, each string contains the value of one of the environment variables, the end of the envp array is marked by a zero entry

* Each entry in the envp array has the following format

```
name = value
```

where name is the name of the environment variable and value is its value

* For example, if the value of the HOME variable is /usr/prof/tony, then its envp array entry would be

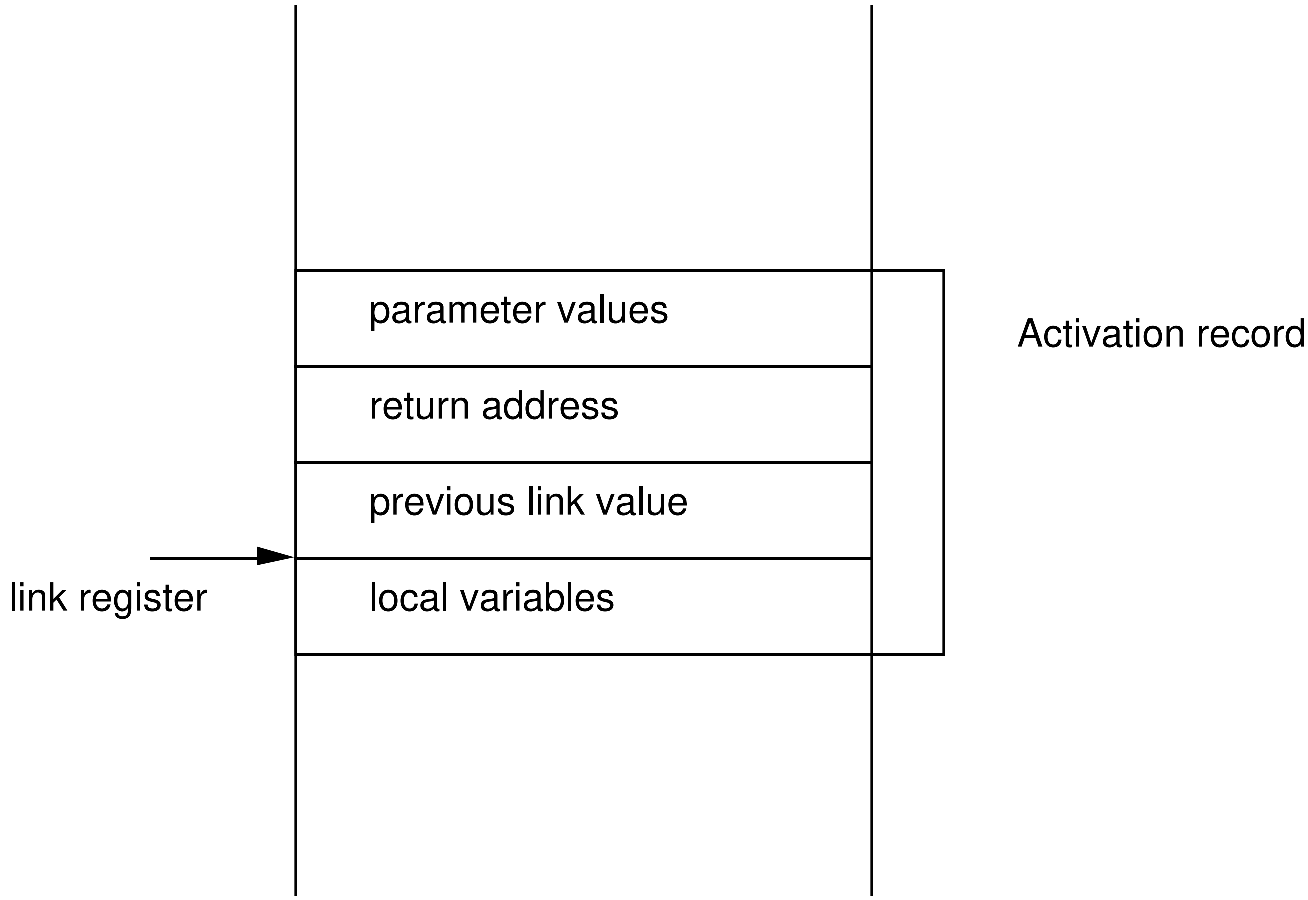
```
envp[i] = "HOME=/usr/prof/tony"
```

* We need to search through this array if we want to find the value of a particular environment variable

* The startup code places the values of argc, argv, and envp on the stack and then transfers control to the main() procedure

Calling a Procedure

* Now lets see what happens when we call a procedure



* The first thing that happens is the calling procedure pushes the parameter values onto the stack

* Next a (subroutine) jump (or similar instruction) to the called procedure is performed, this pushes the return address for the calling procedure onto the stack

* The called procedure then sets up its link register, the previous link register value is pushed onto the stack and the new value is set to the top of the stack

* Next stack space is allocated for the called procedure's local variables, the address of a local variable is relative to the link register

* All of this information that has been put on the stack is called the activation record or stack frame for the procedure - this activation record must be constructed each time a procedure is called

* When the procedure is ready to return all the above operations need to be reversed

* The return value of the procedure is placed in a register, the calling procedure will then retrieve this value after the return

* The link register and the stack pointer are then restored to their previous values and an rts is executed

* Nested procedure calls and recursive procedure calls cause the stack to grow towards the data segment - if you have an infinite recursion the stack will continue to grow until it reaches the data segment --at that point your program will be terminated with a core dump

Storage Classes

*   Storage classes are used for two purposes in C:

*   control the visibility of a name

*   specify how storage is allocated for a variable

*   There are 4 storage classes:

1. auto
2. static
3. extern
4. register

Declarations and Definitions

*   Recall the difference between a declaration and a definition

*   A declaration states that a name exists and how it can be used - a procedure declaration gives the name of the procedure, its return type and optionally the types of its parameters

*   A declaration only tells the compiler that the procedure or variable exists - the compiler doesn't create the procedure or variable

* A definition contains all the information that a declaration does, plus it creates the procedure or variable

* A procedure definition contains the statements in the body of the procedure, the procedure is actually created

* A variable definition allocates memory to store the variable value, the variable is actually created

Visibility and Storage Classes

* A variable or procedure that is declared extern is not defined in the current file, its definition is in another file that will be loaded with the current file

* A procedure is visible outside of the current file by default, that is you can call a procedure defined in another file

* If a procedure is not declared in the current file, but is called by a procedure in the current file, C will provide a default declaration for the procedure, by default it returns an integer value

* If a procedure is declared static, then its definition cannot be seen outside of the current file, a static procedure can only be called by the procedures that are defined in the current file - it is not seen outside of this file

In summary:

1. by default a procedure definition is exported to other files

2. if static is used the procedure definition is local to the current file

* By default, a variable is visible within its current scope, if a variable is defined inside of a procedure it is only visible within that procedure

* If a variable is defined outside of a procedure, a global variable, it is visible to the entire program - it can be seen in the current file and any other file that is loaded with the current file

* If a global variable is defined with the static storage class it can only be seen within the current file - other files cannot reference this variable

* A variable that is declared extern is defined in another file, but is referenced in the current file - C will not automatically provide declaration for external variables like it does for procedures

Summary:

1. local variables can only be seen within their procedure
2. by default global variables can be seen everywhere

3. statically defined global variables can only be seen in the current file
4. external variables must be declared

Storage Allocation and Storage Classes

* By default local variables have the storage class auto, they are allocated on the stack when the procedure is called and their memory is released when the procedure returns - the values of auto variables are not saved between calls

* A static local variable is not allocated on the stack, it is allocated in the data segment, the memory for a static variable is allocated by the compiler, the value of a static variable is saved when its procedure returns, when the procedure is called again it will still have the same value, for example:

```
int count(void) {
    static int calls = 1;

    return(calls++);
}
```

* A register variable is stored in a machine register, if one is available - this is only a hint to the compiler that the variable is heavily used and the procedure may run faster if the variable is stored in a register

Typedef

* typedef can be used to declare a name for a type, this doesn't declare a new type, it just creates a name for an existing type

* A typedef declaration has the following format

```
typedef type type_name ;
```

For example:

```
typedef char* string;
```

* We can now use string in place of a type in a variable declaration:

```
string s;
```

* This allows us to hide the definition of types in include files, can change the type without changing all the programs that use it, it is also a short form for more complex types - not as much to type and remember