P6 – Scientific Programming

Marcus Mohr
Jens Oeser

Geophysics Section
Department of Earth and Environmental Sciences
Ludwig-Maximilians-Universität München

SoSe 2021

Part #13

Functions in C (cont.)

function objects, function types, function pointers, stack frame

Intro: Gauss3 (1/4)

The following Gaussian integration formula is exact for polynomials of degree up to 5 on $[-1, 1]$:

$$\sum_{k=1}^{3} \omega_k f(x_k) \ , \quad x_k \in \left(-\frac{\sqrt{15}}{5}, 0, \frac{\sqrt{15}}{5} \right) \ , \quad \omega_k \in \left(\frac{5}{9}, \frac{8}{9}, \frac{5}{9} \right)$$

For general intervals $[a, b]$ we need to adapt weights and points to

$$\hat{x}_k = \frac{b+a}{2} + \frac{b-a}{2} x_k \ , \quad \hat{\omega}_k = \frac{b+a}{2} \omega_k$$

Intro: Gauss3 (2/4)

```
double gauss3( double a, double b ) {

  double weight[] = { 5.0/9.0, 8.0/9.0, 5.0/9.0 };
  double pos[] = { - 0.2 * sqrt( 15.0 ), 0.0, 0.2 * sqrt( 15.0 ) };

  double fac1 = (b+a)*0.5;
  double fac2 = (b-a)*0.5;

  double integral = 0.0;
  for( int k = 0; k < 3; k++ ) {
    integral += fac2 * weight[k] * func( fac1 + fac2 * pos[k] );
  }

  return integral;
}
```

In order to test our algorithm/implementation we need to implement a function `double func( double );` for the integrand.

Intro: Gauss3 (3/4)

Testing with a degree 3 polynomial:

```
double func( double x ) {
  return x*x*x + 2.0*x*x - 3.0*x + 4.0;
}

int main( void ) {
  double integ =  gauss3( 1.0, 2.0 );
  double exact = 95.0 / 12.0;

  printf( "Gauss3: Value of integral = %f\n", integ );
  printf( "Exact:  Value of integral = %f\n", exact );
  printf( "Error:  %e\n", exact - integ );
}
```

```
Gauss3: Value of integral = 7.916667
Exact:  Value of integral = 7.916667
Error:  8.881784e-16
```

Intro: Gauss3 (4/4)

- Now let us test with a polynomial of degree 5
  $\rightarrow$ need to change implementation of `func()`
- What if we want to run a series of tests in one program?
- We would need a way to pass the integrand function `func()` to the function `gauss3()`

Intro: Gauss3 (4/4)

- Now let us test with a polynomial of degree 5
  $\rightarrow$ need to change implementation of `func()`
- What if we want to run a series of tests in one program?
- We would need a way to pass the integrand function `func()` to the function `gauss3()`
- We can add a formal parameter that is of the function type of the function we want to use.

Variant #1

```
double gauss3( double a, double b, double func( double ) ) {

  double weight[] = { 5.0/9.0, 8.0/9.0, 5.0/9.0 };
  double pos[] = { - 0.2 * sqrt( 15.0 ), 0.0, 0.2 * sqrt( 15.0 ) };

  double fac1 = (b+a)*0.5;
  double fac2 = (b-a)*0.5;

  double integral = 0.0;
  for( int k = 0; k < 3; k++ ) {
    integral += fac2 * weight[k] * func( fac1 + fac2 * pos[k] );
  }

  return integral;
}
```

```c
double poly3( double x ) {
  return x*x*x + 2.0*x*x - 3.0*x + 4.0;
}

double poly5( double x ) {
  return 0.1 * x*x*x*x*x + 2.0*x*x - 3.0*x + 4.0;
}

int main( void ) {
  double integ = 0.0;
  double exact = 0.0;

  // test with 3rd order polynomial
  integ =  gauss3( 1.0, 2.0, poly3 );
  exact = 95.0 / 12.0;
  printf( "Gauss3: Value of integral = %f\n", integ );
  printf( "Exact:  Value of integral = %f\n", exact );
  printf( "Error:  %e\n", exact - integ );

  // test with 5th order polynomial
  integ =  gauss3( 1.0, 2.0, poly5 );
  exact = 313.0 / 60.0;
  printf( "Gauss3: Value of integral = %f\n", integ );
  printf( "Exact:  Value of integral = %f\n", exact );
  printf( "Error:  %e\n", exact - integ );
}
```

Function Addresses

- One of the design principles of the von Neumann architecture was the stored program approach.
- The machine code of a function is stored in the memory of our computer and, thus, has an address.

```
void f( int a ) {
  printf( "a = %d\n", a );
}

int main( void ) {
  int a = 42;
  printf( "address of a = %p\n", &a );
  printf( "address of f = %p\n", &f );
  printf( "address of f = %p\n",  f );
```

When executed the code might print:

```
address of a = 0x7ffea69bac4c
address of f = 0x4004f0
address of f = 0x4004f0
```

Note that `f` and `&f` result in the same.

Function Pointers

- We can also define function pointers, i.e. variables that hold the address of a function of an associated function type.
- Actually in variant #1 that was what our parameter 'decayed' into.

```
void f( int a ) {
  printf( "a = %d\n", a );
}

int main( void ) {
  int a = 42;
  printf( "address of a = %p\n", &a );
  printf( "address of f = %p\n", &f );
  printf( "address of f = %p\n",  f );

  // define a function pointer
  void (*fPtr) ( int ) = NULL;
  printf( "fPtr = %p\n", fPtr );

  fPtr = f;
  printf( "address of f = %p\n", fPtr );
}
```

When executed the code might print:

```
address of a = 0x7fff534937bc
address of f = 0x5623cbf776aa
address of f = 0x5623cbf776aa
fPtr = (nil)
address of f = 0x5623cbf776aa
```

Typedefs

- Using function types or function pointers in interfaces quickly becomes hard to read.
- typedef can help here:

```
// declare a function pointer datatype for our integrand
typedef double (*integ_t) (double);

// alternatively we can use a two-step approach:
// step 1: generate an alias for the function type
typedef double (integrand) (double);
// step 2: now an alias for the function pointer type
typedef integrand* integ_t;
```

- With this we can write the interface as

```
double gauss3( double a, double b, integ_t func );
```

```
double poly3( double x ) {
  return x*x*x + 2.0*x*x - 3.0*x + 4.0;
}

double poly5( double x ) {
  return 0.1 * x*x*x*x*x + 2.0*x*x - 3.0*x + 4.0;
}

double poly6( double x ) {
  return 2 * x*x*x*x*x*x - 32.0;
}

int main( void ) {

  integ_t testFuncs[] = { poly3, poly5, poly6 };
  double testValues[] = { 95.0/12.0, 313.0/60.0, 30.0/7.0 };
  double integ = 0.0;
  double exact = 0.0;

  for( int k = 0; k < 3; k++ ) {
    integ = gauss3( 1.0, 2.0, testFuncs[k] );
    exact = testValues[k];
    printf( "Gauss3: Value of integral = %f\n", integ );
    printf( "Exact:  Value of integral = %f\n", exact );
    printf( "Error:  %e\n", exact - integ );
  }
}
```

Test Results for gauss3

```
// degree 3
Gauss3: Value of integral = 7.916667
Exact:  Value of integral = 7.916667
Error:  8.881784e-16

// degree 5
Gauss3: Value of integral = 5.216667
Exact:  Value of integral = 5.216667
Error:  0.000000e+00

// degree 6
Gauss3: Value of integral = 4.285000
Exact:  Value of integral = 4.285714
Error:  7.142857e-04
```

Function Calls and the Stack

- We are going to take a (generalised) look at what happens when our program calls a function. [details depend on OS & HW]
- The computer keeps track of the currently active functions inside our running program with a (call/execution/program) stack.
- When we compile our program the compiler generates a stack frame for each of our functions.
- When the function is called a stack frame instance is placed on top of the call stack.

Stack Frame (1/6)

- The stack frame is a template for the memory requirements of the function (parameters, local variables, return value, ...)
- Let us take a look at how this works for the following function

```
int sum3( int a, int b, int c ) {
  int sum = 0;
  sum = a + b + c;
  return sum;
}
```

- and its driver

```
int main( void ) {
  int x = 1;
  x = sum3( x, x, x );
  return 0;
}
```

Stack Frame (2/6)

```
int sum3( int a, int b, int c ) {
  int sum = 0;
  sum = a + b + c;
  return sum;
}
```

int	sum		24
void*	pc		16
int	rv		12
int	c		8
int	b		4
int	a		0

- Note that the stack frame is only a template; that's why addresses start at 0.
- We need memory for the three parameters `a, b, c` and the local variable `sum`.
- `rv` is space for storing the return value.
- `pc` (program counter) is the address at which to continue execution once `sum3()` has finished.

Stack Frame (3/6)

```
1 int main( void ) {
2   int x = 1;
3   x = sum3( x, x, x );
4   return 0;
5 }
```

int	x	1	1000
void*	pc	"system"	992
int	rv	$\otimes$	988

- Main itself is also a function, with its own stack frame.
- When we execute it an instance of the frame is placed on the call stack (addresses in example are chosen arbitrarily).
- Starting to execute line #3 stack will have shown values.
- `rv` is still undefined; "system" represents code inserted by the compiler to interface our program with the OS.

Stack Frame (3/6)

```
1 int main( void ) {
2   int x = 1;
3   x = sum3( x, x, x );
4   return 0;
5 }
```

int	x	1	1000
void*	pc	"system"	992
int	rv	⊗	988

- Main itself is also a function, with its own stack frame.
- When we execute it an instance of the frame is placed on the call stack (addresses in example are chosen arbitrarily).
- Starting to execute line #3 stack will have shown values.
- `rv` is still undefined; "system" represents code inserted by the compiler to interface our program with the OS.
- now in line #3 we call `sum3()`

Stack Frame (4/4)

- Call to `sum3()` triggers that an instance of its stack frame is placed on the call stack.
- Happens on top of the frame for `main` using the next addresses.

`int`	`x`	1	1000
`void*`	`pc`	"system"	992
`int`	`rv`	$\otimes$	988

Stack Frame (4/4)

- Call to `sum3()` triggers that an instance of its stack frame is placed on the call stack.
- Happens on top of the frame for `main` using the next addresses.
- First the three parameters are filled with their arguments.

`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void*`	`pc`	“system”	992
`int`	`rv`	$\otimes$	988

Stack Frame (4/4)

- Call to `sum3()` triggers that an instance of its stack frame is placed on the call stack.
- Happens on top of the frame for `main` using the next addresses.
- First the three parameters are filled with their arguments.
- Space for return value and `pc` is allocated; the latter is set.

`int*`	`pc`	"line 3+"	1020
`int`	`rv`	$\otimes$	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void*`	`pc`	"system"	992
`int`	`rv`	$\otimes$	988

Stack Frame (4/4)

- Call to `sum3()` triggers that an instance of its stack frame is placed on the call stack.
- Happens on top of the frame for `main` using the next addresses.
- First the three parameters are filled with their arguments.
- Space for return value and `pc` is allocated; the latter is set.
- Finally space for `sum` is allocated; the code of `sum3` initialises it to 0.

`int`	`sum`	0	1028
`int*`	`pc`	"line 3+"	1020
`int`	`rv`	$\otimes$	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void*`	`pc`	"system"	992
`int`	`rv`	$\otimes$	988

Stack Frame (5/6)

- Next we execute `sum = a + b + c;` changing the value of local variable `sum`.

`int`	`sum`	3	1028
`int*`	`pc`	"line 3+"	1020
`int`	`rv`	$\otimes$	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void*`	`pc`	"system"	992
`int`	`rv`	$\otimes$	988

Stack Frame (5/6)

- Next we execute `sum = a + b + c;` changing the value of local variable `sum`.
- Then we execute `return sum;` which sets the return value `rv`.

`int`	`sum`	3	1028
`int*`	`pc`	"line 3+"	1020
`int`	`rv`	3	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void*`	`pc`	"system"	992
`int`	`rv`	$\otimes$	988

Stack Frame (5/6)

- Next we execute `sum = a + b + c;` changing the value of local variable `sum`.
- Then we execute `return sum;` which sets the return value `rv`.
- Now `sum3` is done and we can start to deconstruct the stack frame instance again.

int	sum	3	1028
int*	pc	"line 3+"	1020
int	rv	3	1016
int	c	1	1012
int	b	1	1008
int	a	1	1004
int	x	1	1000
void*	pc	"system"	992
int	rv	$\otimes$	988

Stack Frame (5/6)

- Next we execute `sum = a + b + c;` changing the value of local variable `sum`.
- Then we execute `return sum;` which sets the return value `rv`.
- Now `sum3` is done and we can start to deconstruct the stack frame instance again.
- This starts with deallocating (popping) the local variables.

`int*`	`pc`	“line 3+”	1020
`int`	`rv`	3	1016
`int`	`c`	1	1012
`int`	`b`	1	1008
`int`	`a`	1	1004
`int`	`x`	1	1000
`void*`	`pc`	“system”	992
`int`	`rv`	$\otimes$	988

Stack Frame (5/6)

- Next we execute `sum = a + b + c;` changing the value of local variable `sum`.
- Then we execute `return sum;` which sets the return value `rv`.
- Now `sum3` is done and we can start to deconstruct the stack frame instance again.
- This starts with deallocating (popping) the local variables.
- Program counter is restored from `pc` and then `pc` is popped.

int	rv	3	1016
int	c	1	1012
int	b	1	1008
int	a	1	1004
int	x	1	1000
void*	pc	“system”	992
int	rv	$\otimes$	988

Stack Frame (6/6)

- Control is now back at line #3 of `main` which reads `x = sum3(x,x,x);`

int	rv	3	1016
int	c	1	1012
int	b	1	1008
int	a	1	1004
int	x	1	1000
void*	pc	"system"	992
int	rv	$\otimes$	988

Stack Frame (6/6)

- Control is now back at line #3 of `main` which reads `x = sum3(x,x,x);`
- The variable `x` (local to main) is updated using `rv` and then `rv` is popped.

int	rv	3	1016
int	c	1	1012
int	b	1	1008
int	a	1	1004
int	x	1	1000
void*	pc	"system"	992
int	rv	⊗	988

Stack Frame (6/6)

- Control is now back at line #3 of `main` which reads `x = sum3(x,x,x);`
- The variable `x` (local to main) is updated using `rv` and then `rv` is popped.
- Finally the arguments are popped.

int	c	1	1012
int	b	1	1008
int	a	1	1004
int	x	3	1000
void*	pc	"system"	992
int	rv	$\otimes$	988

Stack Frame (6/6)

- Control is now back at line #3 of `main` which reads `x = sum3(x,x,x);`
- The variable `x` (local to main) is updated using `rv` and then `rv` is popped.
- Finally the arguments are popped.

`int`	`x`	3	1000
`void*`	`pc`	"system"	992
`int`	`rv`	$\otimes$	988