Memory Alignment and Padding

Prof. Jyotiprakash Mishra
mail@jyotiprakash.org

Basic Type Sizes

Program 1:

```c
#include <stdio.h>
int main() {
  printf("char: %lu byte(s)\n",
    sizeof(char));
  printf("short: %lu byte(s)\n",
    sizeof(short));
  printf("int: %lu byte(s)\n",
    sizeof(int));
  printf("long: %lu byte(s)\n",
    sizeof(long));
  printf("float: %lu byte(s)\n",
    sizeof(float));
  printf("double: %lu byte(s)\n",
    sizeof(double));
  printf("pointer: %lu byte(s)\n",
    sizeof(void*));
  return 0;
}
```

Output (64-bit):

```
char: 1 byte(s)
short: 2 byte(s)
int: 4 byte(s)
long: 8 byte(s)
float: 4 byte(s)
double: 8 byte(s)
pointer: 8 byte(s)
```

Note:

```
sizeof returns size in bytes.
Sizes platform-dependent but
typically: char=1, short=2, int=4,
long=8 (64-bit), pointer=8 (64-bit).
```

Simple Struct Padding

Program 2:

```c
#include <stdio.h>
struct Simple {
  char c;
  int i;
};
int main() {
  struct Simple s;
  printf("sizeof(struct Simple): %lu\n",
    sizeof(struct Simple));
  printf("sizeof(char): %lu\n",
    sizeof(char));
  printf("sizeof(int): %lu\n",
    sizeof(int));
  printf("Sum: %lu\n",
    sizeof(char) + sizeof(int));
  printf("Address of c: %p\n", &s.c);
  printf("Address of i: %p\n", &s.i);
  return 0;
}
```

Output:

```
sizeof(struct Simple): 8
sizeof(char): 1
sizeof(int): 4
Sum: 5
Address of c: 0x7fff...00
Address of i: 0x7fff...04
```

Note:

```
Struct size 8, not 5. Padding added
after char to align int on 4-byte
boundary. 3 bytes padding.
Layout: [c][pad][pad][pad][i][i][i][i]
```

Alignment Requirements

Program 3:

```c
#include <stdio.h>
#include <stddef.h>
struct Example {
  char c;
  int i;
  short s;
};
int main() {
  printf("Size: %lu\n",
    sizeof(struct Example));
  printf("Offset of c: %lu\n",
    offsetof(struct Example, c));
  printf("Offset of i: %lu\n",
    offsetof(struct Example, i));
  printf("Offset of s: %lu\n",
    offsetof(struct Example, s));
  return 0;
}
```

Output:

```
Size: 12
Offset of c: 0
Offset of i: 4
Offset of s: 8
```

Note:

```
offsetof macro shows member offset.
c at 0, 3 bytes padding, i at 4,
s at 8, 2 bytes end padding.
Layout: [c][--][--][--][iiii][ss][--]
```

Reordering Members

Program 4:

```c
#include <stdio.h>
struct Bad {
  char c1;
  int i;
  char c2;
  short s;
};
struct Good {
  int i;
  short s;
  char c1;
  char c2;
};
int main() {
  printf("sizeof(Bad): %lu\n",
    sizeof(struct Bad));
  printf("sizeof(Good): %lu\n",
    sizeof(struct Good));
  return 0;
}
```

Output:

```
sizeof(Bad): 12
sizeof(Good): 8
```

Note:

Order matters! Bad layout wastes space. Good layout groups by size (descending). Bad: 12 bytes with padding. Good: 8 bytes, minimal padding. Same data, different order.

Packed Structs

Program 5:

```c
#include <stdio.h>
struct Normal {
  char c;
  int i;
  short s;
};
struct __attribute__((packed)) Packed {
  char c;
  int i;
  short s;
};
int main() {
  printf("sizeof(Normal): %lu\n",
    sizeof(struct Normal));
  printf("sizeof(Packed): %lu\n",
    sizeof(struct Packed));
  return 0;
}
```

Output:

```
sizeof(Normal): 12
sizeof(Packed): 7
```

Note:

```
packed attribute removes padding.
Normal: 12 bytes with alignment.
Packed: 7 bytes, no padding.
Warning: unaligned access slower,
may crash on some architectures.
```

Pragma Pack

Program 6:

```c
#include <stdio.h>
#pragma pack(push, 1)
struct Packed1 {
  char c;
  int i;
  short s;
};
#pragma pack(pop)
struct Normal {
  char c;
  int i;
  short s;
};
int main() {
  printf("sizeof(Packed1): %lu\n",
    sizeof(struct Packed1));
  printf("sizeof(Normal): %lu\n",
    sizeof(struct Normal));
  return 0;
}
```

Output:

```
sizeof(Packed1): 7
sizeof(Normal): 12
```

Note:

```
#pragma pack(1): 1-byte alignment.
pack(push, 1): save current, set to 1.
pack(pop): restore previous.
Affects all structs between
push and pop.
```

Nested Structs Padding

Program 7:

```c
#include <stdio.h>
struct Inner {
  char c;
  int i;
};
struct Outer {
  char c;
  struct Inner inner;
  short s;
};
int main() {
  printf("sizeof(Inner): %lu\n",
    sizeof(struct Inner));
  printf("sizeof(Outer): %lu\n",
    sizeof(struct Outer));
  struct Outer o;
  printf("Offset of c: %lu\n",
    (char*)&o.c - (char*)&o);
  printf("Offset of inner: %lu\n",
    (char*)&o.inner - (char*)&o);
  return 0;
}
```

Output:

```
sizeof(Inner): 8
sizeof(Outer): 16
Offset of c: 0
Offset of inner: 4
```

Note:

```
Nested struct follows alignment
rules. Inner needs 4-byte alignment,
so 3-byte padding after first c.
Inner takes 8 bytes, s gets 2-byte
pad at end.
```

Array in Struct

Program 8:

```c
#include <stdio.h>
struct WithArray {
  char c;
  int arr[3];
  char d;
};
int main() {
  struct WithArray wa;
  printf("sizeof(WithArray): %lu\n",
    sizeof(struct WithArray));
  printf("Offset of c: %lu\n",
    (char*)&wa.c - (char*)&wa);
  printf("Offset of arr: %lu\n",
    (char*)&wa.arr - (char*)&wa);
  printf("Offset of d: %lu\n",
    (char*)&wa.d - (char*)&wa);
  return 0;
}
```

Output:

```
sizeof(WithArray): 20
Offset of c: 0
Offset of arr: 4
Offset of d: 16
```

Note:

```
Array aligned as its element type.
int array needs 4-byte alignment.
3-byte padding after c. Array takes
12 bytes. d at 16, 3-byte end pad.
```

Union Alignment

Program 9:

```c
#include <stdio.h>
union Data {
  char c;
  int i;
  double d;
};
struct WithUnion {
  char c;
  union Data data;
};
int main() {
  printf("sizeof(union Data): %lu\n",
    sizeof(union Data));
  printf("sizeof(struct WithUnion): %lu\n",
    sizeof(struct WithUnion));
  return 0;
}
```

Output:

```
sizeof(union Data): 8
sizeof(struct WithUnion): 16
```

Note:

```
Union size is largest member.
double is 8 bytes, so union is 8.
Union aligned to strictest member
(double needs 8-byte alignment).
7-byte padding after c in struct.
```

Bit Fields Basics

Program 10:

```c
#include <stdio.h>
struct Flags {
  unsigned int flag1 : 1;
  unsigned int flag2 : 1;
  unsigned int value : 6;
};
int main() {
  struct Flags f;
  printf("sizeof(Flags): %lu\n",
    sizeof(struct Flags));
  f.flag1 = 1;
  f.flag2 = 0;
  f.value = 42;
  printf("flag1: %u\n", f.flag1);
  printf("flag2: %u\n", f.flag2);
  printf("value: %u\n", f.value);
  return 0;
}
```

Output:

```
sizeof(Flags): 4
flag1: 1
flag2: 0
value: 42
```

Note:

```
Bit fields pack multiple fields into
single int. flag1, flag2: 1 bit each.
value: 6 bits. Total 8 bits packed
in 4-byte int. Saves space for flags.
```

Bit Fields Padding

Program 11:

```c
#include <stdio.h>
struct BitPad {
  unsigned char a : 4;
  unsigned char b : 4;
  unsigned char c : 4;
};
struct NoBitPad {
  unsigned int a : 4;
  unsigned int b : 4;
  unsigned int c : 4;
};
int main() {
  printf("sizeof(BitPad): %lu\n",
    sizeof(struct BitPad));
  printf("sizeof(NoBitPad): %lu\n",
    sizeof(struct NoBitPad));
  return 0;
}
```

Output:

```
sizeof(BitPad): 2
sizeof(NoBitPad): 4
```

Note:

```
Bit fields can't span storage units.
BitPad: a,b in first char (8 bits),
c in second char. 2 bytes total.
NoBitPad: all fit in one int. 4 bytes
due to int alignment.
```

Alignment of Arrays

Program 12:

```c
#include <stdio.h>
struct Element {
  char c;
  int i;
};
int main() {
  struct Element arr[3];
  printf("sizeof(Element): %lu\n",
    sizeof(struct Element));
  printf("sizeof(arr): %lu\n",
    sizeof(arr));
  int i;
  for (i = 0; i < 3; i++) {
    printf("arr[%d] at %p\n",
      i, &arr[i]);
  }
  return 0;
}
```

Output:

```
sizeof(Element): 8
sizeof(arr): 24
arr[0] at 0x7fff...00
arr[1] at 0x7fff...08
arr[2] at 0x7fff...10
```

Note:

```
Array elements evenly spaced by
element size. Each Element is 8 bytes
(with padding). Array has no extra
padding between elements.
```

Flexible Array Members

Program 13:

```c
#include <stdio.h>
#include <stdlib.h>
struct Flexible {
  int count;
  int data[];
};
int main() {
  printf("sizeof(Flexible): %lu\n",
    sizeof(struct Flexible));
  struct Flexible *f =
    malloc(sizeof(struct Flexible) +
      5 * sizeof(int));
  f->count = 5;
  int i;
  for (i = 0; i < 5; i++)
    f->data[i] = i * 10;
  for (i = 0; i < f->count; i++)
    printf("%d ", f->data[i]);
  printf("\n");
  free(f);
  return 0;
}
```

Output:

```
sizeof(Flexible): 4
0 10 20 30 40
```

Note:

```
Flexible array member (FAM): array
without size as last member. Size
doesn't include FAM. Allocate extra
space manually. C99 feature.
```

Cache Line Alignment

Program 14:

```c
#include <stdio.h>
struct Aligned64
  __attribute__((aligned(64))) {
  int value;
};
struct Normal {
  int value;
};
int main() {
  struct Aligned64 a1, a2;
  struct Normal n1, n2;
  printf("sizeof(Aligned64): %lu\n",
    sizeof(struct Aligned64));
  printf("&a1: %p, &a2: %p\n", &a1, &a2);
  printf("Distance: %ld\n",
    (char*)&a2 - (char*)&a1);
  printf("sizeof(Normal): %lu\n",
    sizeof(struct Normal));
  return 0;
}
```

Output:

```
sizeof(Aligned64): 64
&a1: 0x7fff...c0, &a2: 0x7fff...80
Distance: -64
sizeof(Normal): 4
```

Note:

```
aligned(64) forces 64-byte alignment.
Useful for cache line optimization.
Cache lines typically 64 bytes.
Prevents false sharing in threading.
```

Zero-Length Arrays

Program 15:

```c
#include <stdio.h>
#include <stdlib.h>
struct ZeroLen {
  int count;
  int data[0];
};
int main() {
  printf("sizeof(ZeroLen): %lu\n",
    sizeof(struct ZeroLen));
  struct ZeroLen *z =
    malloc(sizeof(struct ZeroLen) +
      3 * sizeof(int));
  z->count = 3;
  z->data[0] = 100;
  z->data[1] = 200;
  z->data[2] = 300;
  printf("Data: %d %d %d\n",
    z->data[0], z->data[1], z->data[2]);
  free(z);
  return 0;
}
```

Output:

```
sizeof(ZeroLen): 4
Data: 100 200 300
```

Note:

```
Zero-length array (GNU extension).
Similar to flexible array member.
data[0] takes no space in struct.
Allocate extra for actual data.
Older alternative to FAM.
```

Struct vs Class Alignment

Program 16:

```c
#include <stdio.h>
struct A { char c; };
struct B { char c; double d; };
struct C { double d; char c; };
struct D { char c1; char c2; int i; };
int main() {
  printf("A: %lu\n", sizeof(struct A));
  printf("B: %lu\n", sizeof(struct B));
  printf("C: %lu\n", sizeof(struct C));
  printf("D: %lu\n", sizeof(struct D));
  printf("Expected B: %lu\n",
    sizeof(char) + sizeof(double));
  printf("Expected C: %lu\n",
    sizeof(double) + sizeof(char));
  return 0;
}
```

Output:

```
A: 1
B: 16
C: 16
D: 8
Expected B: 9
Expected C: 9
```

Note:

```
B: 7 bytes padding before d, aligns
to 8. C: same size, order doesn't
matter for total with large types.
D: chars consecutive, then 2-byte pad.
```

Pointer Alignment

Program 17:

```c
#include <stdio.h>
int main() {
  char buffer[20];
  int *p1 = (int*)&buffer[0];
  int *p2 = (int*)&buffer[1];
  int *p3 = (int*)&buffer[4];
  printf("buffer address: %p\n", buffer);
  printf("p1 (offset 0): %p\n", p1);
  printf("p2 (offset 1): %p\n", p2);
  printf("p3 (offset 4): %p\n", p3);
  printf("p1 aligned: %s\n",
    ((size_t)p1 % 4 == 0) ? "yes" : "no");
  printf("p2 aligned: %s\n",
    ((size_t)p2 % 4 == 0) ? "yes" : "no");
  printf("p3 aligned: %s\n",
    ((size_t)p3 % 4 == 0) ? "yes" : "no");
  return 0;
}
```

Output:

```
buffer address: 0x7fff...10
p1 (offset 0): 0x7fff...10
p2 (offset 1): 0x7fff...11
p3 (offset 4): 0x7fff...14
p1 aligned: yes
p2 aligned: no
p3 aligned: yes
```

Note:

```
Unaligned pointer access is undefined
behavior. May crash on ARM/SPARC.
Slower on x86. Check alignment with
modulo. Only use aligned pointers.
```

Struct Hole Detection

Program 18:

```c
#include <stdio.h>
#include <stddef.h>
struct Test {
  char a;
  int b;
  char c;
  double d;
};
int main() {
  printf("Size: %lu\n",
    sizeof(struct Test));
  printf("a at %lu (size %lu)\n",
    offsetof(struct Test, a),
    sizeof(char));
  printf("b at %lu (size %lu)\n",
    offsetof(struct Test, b),
    sizeof(int));
  printf("c at %lu (size %lu)\n",
    offsetof(struct Test, c),
    sizeof(char));
  printf("d at %lu (size %lu)\n",
    offsetof(struct Test, d),
    sizeof(double));
  return 0;
}
```

Output:

```
Size: 24
a at 0 (size 1)
b at 4 (size 4)
c at 8 (size 1)
d at 16 (size 8)
```

Note:

```
Holes (padding): 3 bytes after a,
7 bytes after c. Total actual data:
14 bytes. Total size: 24 bytes.
10 bytes wasted! Reorder to save.
```

Optimal Struct Layout

Program 19:

```c
#include <stdio.h>
struct Unoptimized {
  char a;
  double d;
  char b;
  int i;
  char c;
};
struct Optimized {
  double d;
  int i;
  char a;
  char b;
  char c;
};
int main() {
  printf("Unoptimized: %lu bytes\n",
    sizeof(struct Unoptimized));
  printf("Optimized: %lu bytes\n",
    sizeof(struct Optimized));
  printf("Savings: %lu bytes\n",
    sizeof(struct Unoptimized) -
    sizeof(struct Optimized));
  return 0;
}
```

Output:

```
Unoptimized: 32 bytes
Optimized: 16 bytes
Savings: 16 bytes
```

Note:

```
Order by size descending: largest
types first. Unoptimized has many
holes. Optimized groups small types
together. 50% space savings!
```

Real World Example

Program 20:

```c
#include <stdio.h>
struct NetworkPacket {
  unsigned char version : 4;
  unsigned char type : 4;
  unsigned char flags;
  unsigned short length;
  unsigned int timestamp;
  unsigned long long id;
  char data[8];
} __attribute__((packed));
int main() {
  struct NetworkPacket pkt = {
    .version = 1,
    .type = 5,
    .flags = 0x80,
    .length = 100,
    .timestamp = 12345,
    .id = 99999,
    .data = "PAYLOAD"
  };
  printf("Packet size: %lu\n",
    sizeof(struct NetworkPacket));
  printf("v:%u t:%u f:0x%X l:%u t:%u\n",
    pkt.version, pkt.type, pkt.flags,
    pkt.length, pkt.timestamp);
  return 0;
}
```

Output:

```
Packet size: 22
v:1 t:5 f:0x80 l:100 t:12345
```

Note:

```
Network protocol struct. Packed to
match wire format exactly. Bit fields
for version/type. Total 22 bytes.
Without pack would be larger with
padding.
```