ECE 2400 Computer Systems Programming
Fall 2021

Topic 3: C Types

School of Electrical and Computer Engineering
Cornell University

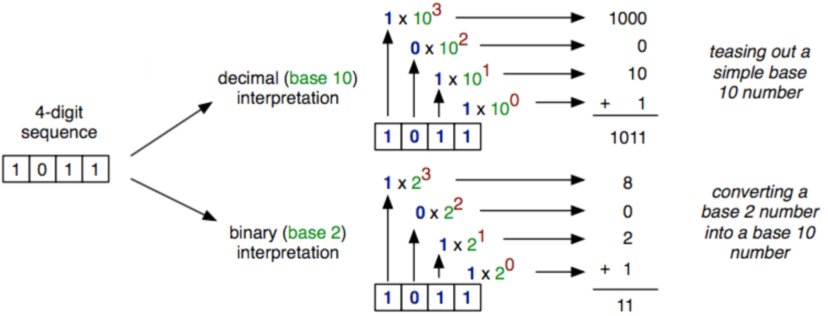
revision: 2021-08-29-22-42

zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

- In C, the type of a variable specifies the kind of values that can be stored in that variable; must answer three questions:
  - What is the meaning of the variable's value?
  - How should the variable's value be stored in the computer
  - What operations are allowed on the variable?
- Critical to keep concept of types separate from concept of values
- C is a statically typed language, meaning that the type of a variable must be known at compile time
- Keep in mind that no matter how complex the type, everything is ultimately stored as a binary number in the computer

1. Binary and Hexadecimal Numbers

Let's review decimal, binary, and hexadecimal number representations.



2. Basic Data Types

We will primarily use the following primitive C types

- `int`: For representing integer numbers
- `char`: For representing characters
- `float` and double: For representing real numbers
- `const T`: For representing constant values of type `T`
- `void`: For representing situations where a value is not allowed

zyBooks The course zyBook includes more information on two's complement representation and the `float`/`double` types for representing real numbers.

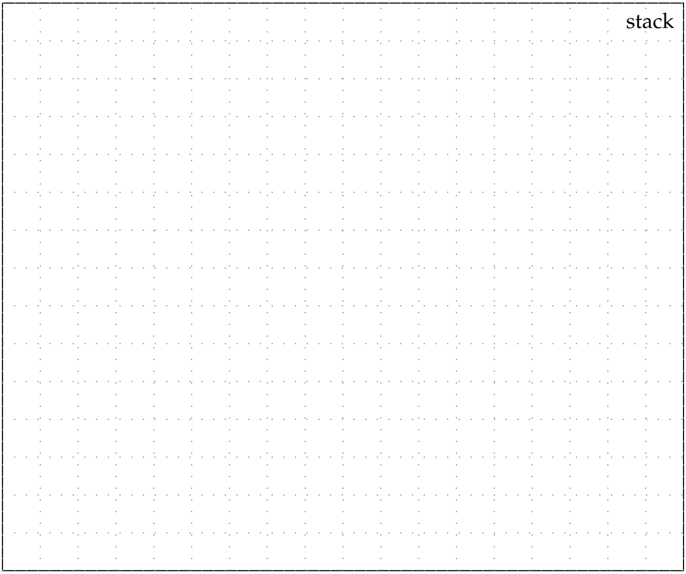
2.1. `int` Type

Meaning?	Integer whole numbers in a limited range
Stored?	32-bit two's complement binary representation
Operations?	Basic integer arithmetic

- Unlike some productivity-level programming languages, variables of type `int` cannot represent arbitrarily large or small integers

- Such variables have a fixed upper limit and lower limit

```c
int avg( int x, int y )
{
  int sum = x + y;
  return sum / 2;
}

int main()
{
  int a = 10;
  int b = 20;
  int c = avg( a, b );
  return 0;
}
```

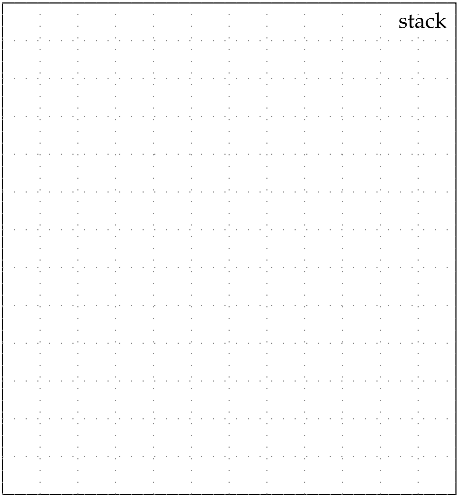


4-Bit Unsigned Integers

- By default, an `int` is short-hand for the type `signed int` which can represent both positive and negative integers
- `unsigned int` can only represent positive integers
- To start, let's focus on variables of type `unsigned int` and let's assume all variables are only four bits

Bits	Unsigned
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

```c
unsigned int a = 4;
unsigned int b = 15;
unsigned int c = 0;
unsigned int d = a + 1;
unsigned int e = b + 1;
unsigned int f = c - 1;
```

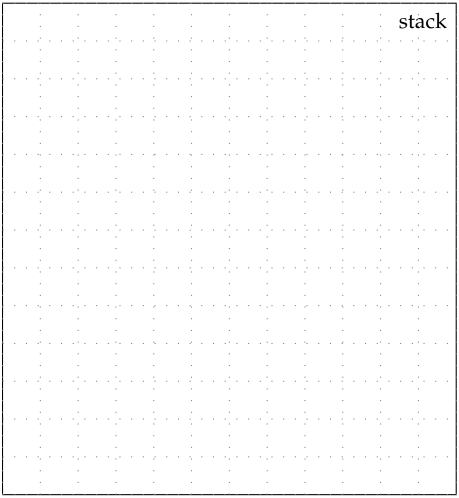


4-Bit Signed Integers

- Now let's consider variables of type `signed int` and let's continue to assume all variables are only four bits
- There can be multiple ways to encode a given value into a sequence of bits (e.g., sign magnitude, one's complement, two's complement)
- The C language specification does not actually specify the exact encoding, but essentially all machines use two's complement

Bits	Sign Mag	One's Comp	Two's Comp
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	–0	–7	–8
1001	–1	–6	–7
1010	–2	–5	–6
1011	–3	–4	–5
1100	–4	–3	–4
1101	–5	–2	–3
1110	–6	–1	–2
1111	–7	–0	–1

```
☐☐☐ signed int a = 4;
☐☐☐ signed int b = 7;
☐☐☐ signed int c = -8;
☐☐☐ signed int d = a + 1;
☐☐☐ signed int e = b + 1;
☐☐☐ signed int f = c - 1;
```



- An `int` is a signed 32-bit binary number
- Can store values between -2,147,483,648 to 2,147,483,647
- What happens if you add one to 2,147,483,647?
- What happens if you subtract one from -2,147,483,648?

- An `unsigned int` is an unsigned 32-bit binary number
- Can store values from 0 to 4,294,967,295
- What happens if you add one to 4,294,967,295?
- What happens if you subtract one from 0?

```c
#include <stdio.h>

int main()
{
  int a = 2147483647;
  int b = a + 1;
  printf("%d + 1 = %d (%x)\n",a,b,b);

  int c = -2147483648;
  int d = c - 1;
  printf("%d - 1 = %d (%x)\n",c,d,d);

  unsigned int e = 4294967295;
  unsigned int f = e + 1;
  printf("%u + 1 = %u (%x)\n",e,f,f);

  unsigned int g = 0;
  unsigned int h = g - 1;
  printf("%u - 1 = %u (%x)\n",g,h,h);

  return 0;
}
```

`https://repl.it/@cbatten/ece2400-T03-ex1`

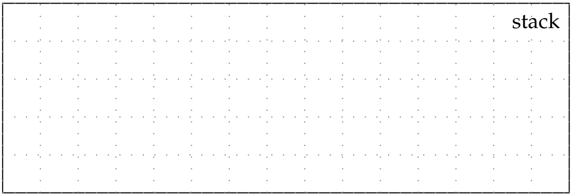
- New format specifiers for hexadecimal (%x) and `unsigned int` (%u)

2.2. `char` Type

- **Meaning?** Character in a “word”
- **Stored?** 8-bit binary representation using ASCII standard
- **Operations?** Basic integer arithmetic

40	(	50	2	60	<	70	F	80	P	90	Z	100	d	110	n
41	)	51	3	61	=	71	G	81	Q	91	[	101	e	111	o
42	*	52	4	62	>	72	H	82	R	92	\	102	f	112	p
43	+	53	5	63	?	73	I	83	S	93	]	103	g	113	q
44	,	54	6	64	@	74	J	84	T	94	^	104	h	114	r
45	-	55	7	65	A	75	K	85	U	95	_	105	i	115	s
46	.	56	8	66	B	76	L	86	V	96	`	106	j	116	t
47	/	57	9	67	C	77	M	87	W	97	a	107	k	117	u
48	0	58	:	68	D	78	N	88	X	98	b	108	l	118	v
49	1	59	;	69	E	79	O	89	Y	99	c	109	m	119	w

```
☐☐☐ 01 char a = 'e';
☐☐☐ 02 char b = 'c';
☐☐☐ 03 char c = 'e';
```



```c
#include <stdio.h>

int main()
{
    char a = 'e';
    char b = 'c';
    char c = 'e';
    printf("%c%c%c\n",a,b,c);
    return 0;
}
```

- New format specifier for `char` (`%c`)

2.3. `const` Types

Meaning?	Indicates variable will not change
Stored?	Whatever is required for "base" type
Operations?	Read-only operations, cannot modify variable

```
// Constant at global scope
const double PI = 3.1415926535;

int main()
{
  const double a = 2.0;
  int b = a * PI;

  const int d = 15;
  d = b;                    // compile time error!

  return 0;
}
```

2.4. `void` Types

Meaning?	No values are allowed
Stored?	No storage needed
Operations?	None

```
void print_line( void )
{
  for ( int i = 0; i < 74; i++ )
    printf("-");
}
```

- Technically, we should use `void` for empty parameter lists
- This applies to `main` as well

3. Programmer-Defined Types

In addition to the default types that are included as part of the C programming language (e.g., `int`, `unsigned int`, `char`, `float`, `double`), C also enables programmers to define their own new types.

zyBooks The course zyBook includes more information on `enum` which enables creating multiple named constants.

3.1. Typedefs

- A typedef actually does *not* define a new type
- A typedef simply provides a new alias for an already defined type

```
typedef type_name new_type_name;
```

- The following code is perfectly fine

```
typedef unsigned int uint_t;
uint_t a = 2;
uint_t b = 3;
unsigned int c = a + b;
```

3.2. `struct` Types

- A `struct` enables bundling multiple variables into a single entity
- A struct definition creates a new type and specifies the type and names of the variables (fields) contained within the `struct`

```
struct _complex_t
{
  double real;
  double imag;
};
typedef struct _complex_t complex_t;
```

```
typedef struct
{
  double real;
  double imag;
}
complex_t;
```

- Struct definitions are at global scope just like function definitions
- Can declare a struct variable just like any other variable

```
complex_t a;      // variable of type complex_t
```

- Structs require a new operator to access the fields
- The dot operator (`.`) composes a struct variable name with a struct field name to create a fully qualified variable name

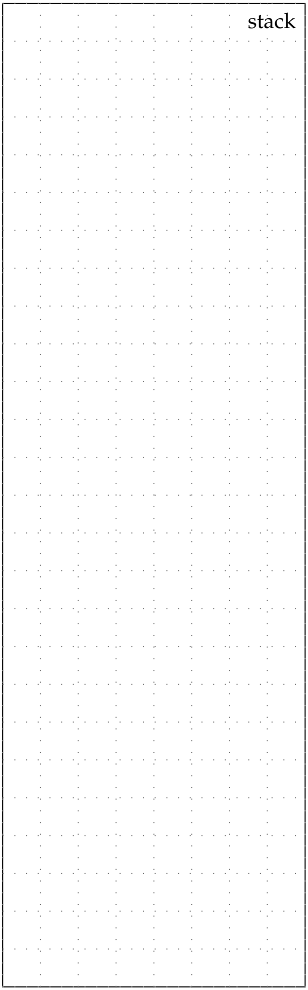
```
complex_t a;      // variable of type complex_t
a.real = 1.0;     // use dot operator to access real field
a.imag = 2.5;     // use dot operator to access imag field

complex_t b;      // variable of type complex_t
b.real = a.real; // use dot operator to copy real field
b.imag = a.imag; // use dot operator to copy imag field
```

- Can copy an entire struct in a single statement
- Semantics of such a copy are to simply copy each field individually

```
complex_t a;      // variable of type complex_t
a.real = 1.0;     // use dot operator to access real field
a.imag = 2.5;     // use dot operator to access imag field

complex_t b;      // variable of type complex_t
b = a;            // copy all fields from a to b
```

- Struct declaration statement simply creates multiple variables on the stack in a single statement

```
typedef struct
{
  int x;
  int y;
}
point_t;

point_t point_add( point_t pt1,
                   point_t pt2 )
{
  point_t pt3;
  pt3.x = pt1.x + pt2.x;
  pt3.y = pt1.y + pt2.y;
  return pt3;
}

int main( void )
{
  point_t pt_a;
  pt_a.x = 2;
  pt_a.y = 3;

  point_t pt_b;
  pt_b.x = 4;
  pt_b.y = 5;

  point_t pt_c;
  pt_c = point_add( pt_a, pt_b );

  return 0;
}
```



4. Working With Types

Types can offer strong static guarantees about correctness, but also need to be carefully managed.

4.1. Type Checking

- Compiler will check to ensure types are consistent
- Inconsistent types will cause a compile-time error

```
typedef struct
{
  int x;
  int y;
}
point_t;

point_t point_add( point_t pt1,
                   point_t pt2 )
{
  point_t pt3;
  pt3.x = pt1.x + pt2.x;
  pt3.y = pt1.y + pt2.y;
  return pt3;
}

int main( void )
{
  int a = 2;
  int b = 3;
  point_t pt_c = point_add( a, b );
  return 0;
}
```

`https://repl.it/@cbatten/ece2400-T03-ex3`

4.2. Type Inference

- Compiler uses type inference to determine type of an expression

```
int   a = 2;
int   b = 3;
int   c = a + b;        // expr (a + b) has type int
int   d = a / b;        // expr (a / b) has type int

float e = 2.0;
float f = 3.0;
float g = e + f;        // expr (e + f) has type float
float h = e / f;        // expr (e / f) has type float
```

4.3. Type Casting

- Type checking prevents assigning a value with a given type `T` to a variable with a different type
- Programmers can use type casting to explicitly convert a value of one type to a value of another type
- The cast operator can be used for explicit type casting

```
unsigned int a = 1;
signed   int b = (signed int) a;

signed   int a = 1;
unsigned int b = (unsigned int) a;

int     a = 2;
float   b = (float) a;

float   a = 2.0;
int     b = (int) a;
```

```
float  a = 2.5;
double b = (double) a;

double a = 2.5;
float  b = (float) a;
```

- Example of using explicit type casting

```
#include <stdio.h>

float avg( int x, int y )
{
  int sum = x + y;
  return ((float) sum) / ((float) 2);
}

int main( void )
{
  float a = 10;
  float b = 15;
  float c = avg( a, b );
  printf(" average of %f and %f is %f\n", a, b, c );
  return 0;
}
```

`https://repl.it/@cbatten/ece2400-T03-ex5`

4.4. Type Conversion

- Compiler can also use implicit type conversion
- Compiler can automatically convert types so they match
- Lower precision types can be converted to higher precision types
- Higher precision types can be converted to lower precision types

```
unsigned int a = 1;
signed   int b = a;

signed   int a = 1;
unsigned int b = a;

int    a = 2;
float  b = a;

float  a = 2.0;
int    b = a;

float  a = 2.5;
double b = a;

double a = 2.5;
float  b = a;

int    a = 2;
float  b = 3;
float  c = a + b;        // converts a to float
float  d = a / b;        // converts a to float

unsigned int a = 2;
signed   int b = -3;
unsigned int c = a * b; // converts a to signed int
```

- Type conversion seems convenient but is at the heart of why C only supports weak static typing and can lead to many subtle bugs

```
unsigned int a = 4294967295;
signed   int b = a;         // careful! b == -1

signed   int a = -1;
unsigned int b = a;         // careful! b == 4294967295

float  a = 2.5;
int    b = a;               // careful! b == 2

double a = 3.14159265358;
float  b = a;               // careful! b != 3.14159265358
```

- The following example illustrates automatic type conversion

```
#include <stdio.h>

int avg( int x, int y )
{
  int sum = x + y;
  return sum / 2;
}

int main( void )
{
  float a = 10;
  float b = 15;
  float c = avg( a, b );
  printf(" average of %f and %f is %f\n", a, b, c );
  return 0;
}
```

`https://repl.it/@cbatten/ece2400-T03-ex4`