

Systemy komputerowe: architektura i programowanie (SYKOM)

Wykład: Język C/C++

Aleksander Pruszkowski

Instytut Telekomunikacji Politechniki Warszawskiej

PLAN WYKŁADY

- Język C/C++ podstawy budowy programów
- Typy danych języka C/C++
 - typy proste
 - złożone typy danych

Język C/C++ podstawy budowy programów

(preprocesor, prototypy funkcji, podział kodu na moduły i narzędzia wsparcia kompilacji)

Język C/C++ podstawy budowy programów

- Znamy listę instrukcji CPU - ale czy trzeba programować w asemblerze?
 - Jak dzisiaj tworzy się oprogramowanie współpracujące ze sprzętem?
 - Zasadniczo w C i coraz częściej w C++ (np.: Arduino)
 - Po co asembler?
 - Podstawa dla twórców kompilatorów
 - Gdy coś w kodzie działa nie tak
 - Gdy fragmenty kodu wygenerowanego z C/C++ działają za wolno
 - Gdy kod ma mieścić się w bardzo małych pamięciach
 - zagadnienie o źródłach ekonomicznych, MCU z wyliczeń finansowych ma kosztować mniej niż 1% kosztów produkcji całego urządzenia (zabawki, gadżety, ...)
 - Gdy chcemy sprawić aby nikt nie był w stanie zrozumieć kodu (;->>)

Język C/C++ podstawy budowy programów

■ Jak wygląda proces kompilacji

■ Przykładowy plik o nazwie main.c:

```
#include <stdio.h>

int main(int argc, char *argv[]){
    printf("Hello world!\n");
    return 0;
}
```

■ Kompilacja: krok 1 – „pre - processing”

- Zasadniczo proces uruchamiany automatycznie
- Dla kompilatora GCC, ręcznie uruchamiany poleceniem:

```
gcc -E main.c > main.E
```

```
main.E:
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "main.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
...
int main(int argc, char *argv[]){
    printf("Hello world!\n");
    return 0;
}
```

Przypomnienie:

- Budowa programu
- Budowa funkcji
- Argumenty wywołania funkcji
- Definicja, deklaracja i inicjacja zmiennej
- Zasięg zmiennych, "życie zmiennych"

Język C/C++ podstawy budowy programów

■ Jak wygląda proces kompilacji

■ Krok 2 - właściwa kompilacja C do ASM

- Także realizowana automatycznie, można podejrzec:

```
gcc -save-temps main.c
```

```
main.c:  
#include <stdio.h>  
int main(int argc, char *argv[]){  
    printf("Hello world!\n");  
    return 0;  
}
```

```
main.s:  
.file "main.c"  
.text  
.section .rodata  
.LC0:  
.string "Hello world!"  
.text  
.globl main  
.type main, @function  
main:  
.LFB0:  
.cfi_startproc  
pushq %rbp  
.cfi_def_cfa_offset 16  
.cfi_offset 6, -16  
movq %rsp, %rbp  
.cfi_def_cfa_register 6  
subq $16, %rsp  
movl %edi, -4(%rbp)  
movq %rsi, -16(%rbp)  
leaq .LC0(%rip), %rdi  
call puts@PLT`
```

Optymalizacja -
tu miał być funkcja printf!

```
movl $0, %eax  
leave  
.cfi_def_cfa 7, 8  
ret  
.cfi_endproc  
.LFE0:  
.size main, .-main  
.ident "GCC: (Debian 8.3.0-6) 8.3.0"  
.section .note.GNU-stack,"",@progbits
```

Język C/C++ podstawy budowy programów

```
main.c:
#include <stdio.h>
int main(int argc, char *argv[]){
    printf("Hello world!\n");
    return 0;
}
```

■ Jak wygląda proces kompilacji

■ Krok 3 - właściwa kompilacja z ASM/C do OBJ (postać pośrednia)

■ Polecenia:

```
gcc -c main.c -o main.o    //kompilacja z C do OBJ
objdump -S main.o          //"Dump" z OBJ do postaci czytelnej dla człowieka
```

```
main.o:      file format elf64-x86-64
```

Disassembly of section .text:

0000000000000000 <main>:

```
0:  55
1:  48 89 e5
4:  48 83 ec 10
8:  89 7d fc
b:  48 89 75 f0
f:  48 8d 3d 00 00 00 00
16: e8 00 00 00 00
1b: b8 00 00 00 00
20: c9
21: c3
```

W tej fazie adres jest
jeszcze nie znany

```
push    %rbp
mov     %rsp,%rbp
sub     $0x10,%rsp
mov     %edi,-0x4(%rbp)
mov     %rsi,-0x10(%rbp)
lea     0x0(%rip),%rdi    # 16 <main+0x16>
callq  1b <main+0x1b>
mov     $0x0,%eax
leaveq
retq
```

Adres symboliczny

≡ call tyle, że dla jest to
instrukcja traktowana jako 64 bitowa
(np.: używa 'rip' zamiast 'eip')

Język C/C++ podstawy budowy programów

```
main.c:
#include <stdio.h>
int main(int argc, char *argv[]){
    printf("Hello world!\n");
    return 0;
}
```

■ Jak wygląda proces kompilacji

■ Krok 4 – linkowanie

■ Polecenia:

```
gcc main.c -o main
```

//Linkowanie – wynik plik wykonywalny „main”

```
objdump -DxS main
```

///"Dump" pliku wykonywalnego do postaci czytelnej dla człowieka

```
main:      file format elf64-x86-64
```

```
main
```

```
architecture: i386:x86-64, flags 0x00000150: HAS_SYMS, DYNAMIC, D_PAGED
```

```
start address 0x00000000000001050
```

```
Program Header:
```

```
...
```

```
Dynamic Section:
```

```
NEEDED      libc.so.6
INIT        0x00000000000001000
```

```
...
```

```
Sections:
```

Idx	Name	Size	VMA	LMA	File off	Algn
...						
13	.text	00000171	00000000000001050	00000000000001050	00001050	2**4
	CONTENTS, ALLOC, LOAD, READONLY, CODE					
23	.data	00000010	00000000000004020	00000000000004020	00003020	2**3
	CONTENTS, ALLOC, LOAD, DATA					
24	.bss	00000008	00000000000004030	00000000000004030	00003030	2**0
	ALLOC					

```
00000000000001135 <main>:
```

```
1135:      55
```

```
...
```

```
1144:      48 8d 3d b9 0e 00 00
```

```
114b:      e8 e0 fe ff ff
```

Teraz call ma finalny adres

```
push    %rbp
```

```
lea     0xeb9(%rip),%rdi # 2004 <_IO_stdin_used+0x4>
```

```
callq   1030 <puts@plt>
```


Język C/C++ podstawy budowy programów

■ Jak wygląda proces kompilacji, cd.

- Czy kompilator to czarna skrzynka?
 - Jeżeli czegoś nie można znaleźć w dokumentacji to dla kompilatorów o otwartych źródłach można przeglądając ich kod i odkryć budowę – zdanie trudne, ale realizowalne!
- Jakie informacje o budowie kompilatora GCC można odkryć automatycznie
 - jak dowiedzieć się gdzie kompilator GCC zagląda podczas swojej pracy, polecenie:

gcc --print-search-dirs

```
install: /usr/lib/gcc/x86_64-linux-gnu/4.9/
programs: =/usr/lib/gcc/x86_64-linux-gnu/4.9/:/usr/lib/gcc/x86_64-linux-
gnu/4.9/:/usr/lib/gcc/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/4.9/:/usr/lib/gcc/x86_64-linux-gnu/:...
libraries: =/usr/lib/gcc/x86_64-linux-gnu/4.9/:/usr/lib/gcc/x86_64-linux-
gnu/4.9/../../../../x86_64-linux-gnu/lib/x86_64-linux-gnu/4.9/:/usr/lib/gcc/x86_64-
linux-gnu/4.9/../../../../x86_64-linux-gnu/lib/x86_64-linux-gnu/:/usr/lib/gcc/x86_64-
linux-gnu/4.9/../../../../x86_64-linux-gnu/lib/../../lib/:/usr/lib/gcc/x86_64-linux-
gnu/4.9/../../../../x86_64-linux-gnu/4.9/:/usr/lib/gcc/x86_64-linux-
gnu/4.9/../../../../x86_64-linux-gnu/:/usr/lib/gcc/x86_64-linux-
gnu/4.9/../../../../lib/:/lib/x86_64-linux-gnu/4.9/:...
```

Język C/C++ podstawy budowy programów

■ Jak wygląda proces kompilacji, cd.

- Jakie informacje o budowie kompilatora GCC można odkryć automatycznie, cd.
 - jak dowiedzieć się z jakich plików bibliotecznych korzysta GCC podczas kompilacji i linkowania, polecenie:

gcc -Wl,-t

```
/usr/bin/ld: mode elf_x86_64
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../x86_64-linux-gnu/crt1.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../x86_64-linux-gnu/crti.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/crtbegin.o
-lgcc_s (/usr/lib/gcc/x86_64-linux-gnu/4.9/libgcc_s.so)
/lib/x86_64-linux-gnu/libc.so.6
(/usr/lib/x86_64-linux-gnu/libc_nonshared.a)elf-init.oS
/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
-lgcc_s (/usr/lib/gcc/x86_64-linux-gnu/4.9/libgcc_s.so)
/usr/lib/gcc/x86_64-linux-gnu/4.9/crtend.o
/usr/lib/gcc/x86_64-linux-gnu/4.9/../../../../x86_64-linux-gnu/crtn.o
```

Język C/C++ podstawy budowy programów

■ Jak wygląda proces kompilacji, cd.

- Jakie informacje o budowie kompilatora GCC można odkryć
 - z jakich makr wbudowanych korzysta GCC, polecenie:

```
gcc -E -dM - < /dev/null
```

```
...
#define __x86_64__ 1
#define __linux__ 1
#define __VERSION__ "4.9.2"
#define __amd64__ 1
...
#define __INT8_MAX__ 127
#define __UINT8_MAX__ 255
#define __INT16_MAX__ 32767
#define __UINT16_MAX__ 65535
#define __UINT32_MAX__ 4294967295U
#define __INT64_MAX__ 9223372036854775807L
#define __UINT64_MAX__ 18446744073709551615UL
#define __INT_MAX__ 2147483647
#define __LONG_MAX__ 9223372036854775807L
#define __LONG_LONG_MAX__ 9223372036854775807LL
...
#define __SIZEOF_SHORT__ 2
#define __SIZEOF_INT__ 4
```

```
#define __SIZEOF_POINTER__ 8
#define __SIZEOF_LONG__ 8
#define __SIZEOF_LONG_LONG__ 8
#define __SIZEOF_FLOAT__ 4
#define __SIZEOF_DOUBLE__ 8
#define __SIZEOF_LONG_DOUBLE__ 16
...
#define __SIZEOF_SIZE_T__ 8
...
#define __INT8_TYPE__ signed char
#define __UINT8_TYPE__ unsigned char
#define __INT16_TYPE__ short int
#define __UINT16_TYPE__ short unsigned int
#define __INT32_TYPE__ int
#define __UINT32_TYPE__ unsigned int
#define __UINT64_TYPE__ long int
#define __UINT64_TYPE__ long unsigned int
...
```

Język C/C++ podstawy budowy programów

■ Linker

- Łączy wszystkie elementy oprogramowania w pojedynczy plik wykonywalny
- Działa bazując się na schemacie zapisanych w pliku z rozszerzeniem „.ld”
 - Opisuje proces łączenia segmentów kodu i danych podczas linkowania
- Kto tworzy pliki „.ld”?
 - Twórca kompilatora
 - Generyczny format – nie dostosowany do pracy z konkretnym sprzętem
 - Twórca/programista sprzętu
 - Zadanie nieco skomplikowane, pomocne jest wywołanie (znane z zajęć lab)
`riscv32-unknown-elf-ld --verbose >sykt.ld`
 - Wygenerowana postać jest dość zawiła – przeznaczona typowych aplikacji
- Czy można zatem utworzyć własny skrypt linkera - tak

Język C/C++ podstawy budowy programów

■ Linker, cd.

■ Format – na przykładzie prostego skryptu

```
ENTRY(Reset_Handler)
MEMORY{
  FLASH(rx) : ORIGIN=0x08000000, LENGTH=16K
  SRAM(rwx) : ORIGIN=0x20000000, LENGTH=2K
}
SECTIONS{
  PROVIDE( __stack_top = ORIGIN(SRAM) + LENGTH(SRAM) );
  .text :
  {
    *(.isr_vector)
    *(.text)
    *(.text.*)
    *(.init)
    *(.fini)
    *(.rodata)
    *(.rodata.*)
    . = ALIGN(4);
  }
  _etext = .;
  > FLASH
```

1) Opis procedury/części kodu od którego program zacznie działanie, na ogół nie jest nią „main()” w lab.1 jest nią „_start:”
zapisany w pliku crt0.s – linker od tej funkcji/procedury zaczyna rozmieszczać kod
2) z reguły CPU zaczyna wykonywać kod spod tego miejsca po RESET

W ramach lab.1 wpisywalibyśmy, np.:

RAM(rwx) : ORIGIN=0X81234560, LENGTH=128M

Aby stos mógł działać poprawnie, w linkerze pojawia się właściwy symbol: __stack_top

Sekcja „text” zawiera właściwy kod programu z wszystkich łączonych plików „.o”

ALIGN(4)-
rozmieszczaj
kod co 4B

*(.isr_vector) – tablica wektorów przerwań

*(.text) i *(.text.*) – sekcje kodu wszystkich łączonych części z kodem wynikowym

*(.init) | *(.fini) – kod inicjacji | zakończenia aplikacji

*(.rodata) i *(.rodata.*) – dane stałe w kodzie

Opisy wskazujące gdzie trafi sekcja zwana „.text”

„>FLASH” oznacza: „zapisz” do tzw. wynikowego „firmware” – czyli obrazu zapisywanego w pamięci trwałej

Język C/C++ podstawy budowy programów

■ Linker, cd.

■ Format – na przykładzie prostego skryptu

Twór pozwalający zorientować się gdzie kończy się pamięć kodu a zaczyna następna sekcja – „.data”

```
_la_data = LOADADDR(.data);
```

```
.data :
```

```
{
    _sdata = .;
    *(.data)
    . = ALIGN(4);
    _edata = .;
```

```
}> SRAM AT> FLASH
```

```
.bss :
```

```
{
    _sbss = .;
    __bss_start__ = _sbss;
    *(.bss)
    *(COMMON)
    . = ALIGN(4);
    _ebss = .;
    __bss_end__ = _ebss;
    . = ALIGN(4);
    end = .;
    __end__ = .;
```

```
}> SRAM
```

```
}
```

Obszar danych inicjowanych w kodzie, np.:

```
int i=12;
void main(){...
```

„>SRAM AT>FLASH” – zapisz podczas działania do pamięci SRAM, a zawartość przenoszona będzie w „firmware”

Obszar globalnych danych niezainicjowanych – tutaj kod oznaczony jako `Reset_Handler`, wyglądający, np.:

```
void Reset_Handler(void) {
```

```
uint32_t size=(uint32_t)&_edata-(uint32_t)&_sdata;
uint8_t *pDst=(uint8_t*)&_sdata; //początek w SRAM
uint8_t *pSrc=(uint8_t*)&_la_data; //początek w flash
for(uint32_t i=0; i<size; i++)
    *pDst++ = *pSrc++; //kopiuj 'data' to SRAM
```

```
size=(uint32_t)&_ebss-(uint32_t)&_sbss;
pDst=(uint8_t*)&_sbss;
for(uint32_t i=0; i<size; i++)
    *pDst++ = 0; //zerowanie sekcji .bss
```

```
main();
```

```
//wywołaj funkcję main(!)
```

```
}
```

„>SRAM” – zapisz (lub lepiej) rozmieść podczas działania w pamięci SRAM, ale treści nie ustalaj

Język C/C++ podstawy budowy programów

■ Jak usprawnić proces kompilacji

- Jak profesjonalnie tworzyć kod z wykorzystaniem GCC i innych narzędzi GNU
 - Istnieje mnóstwo pakietów IDE (Integrated Development Environment)
 - Eclipse, CodeBlocks, Visual Studio,
- W każdym przypadku zaleca się automatyzację procesu kompilacji
 - Istnieje zestaw narzędzi ułatwiających tworzenie plików wynikowych
 - Make, CMake, Autoconf, Automake, Ant(java), ...
- Jak działa Make
 - narzędzie szuka w katalogu wywołania plików Makefile, ...
 - następnie dokonuje kompilacje zgodnie z zapisanymi w tym pliku receptami

Język C/C++ podstawy budowy programów

■ Jak usprawnić proces kompilacji, cd.

■ Przykład prostego pliku Makefile

```
CC=gcc
CFLAGS=-Iinc

all: main.c
    $(CC) $(CFLAGS) main.c -o aplikacja

clean:
    rm -f main.o aplikacja
```

Diagram illustrating the components of a Makefile rule:

- cel** (target): `all:`
- zależność** (prerequisite): `main.c`
- recepta** (recipe): `$(CC) $(CFLAGS) main.c -o aplikacja`

■ Bardziej zaawansowany przykład - rekompilacja selektywna

```
CC=gcc
CFLAGS=-Iinc

all: aplikacja
aplikacja: main.o func.o
    $(CC) main.o func.o -o aplikacja
main.o: main.c
    $(CC) $(CFLAGS) -c main.c -o main.o
func.o: func.c
    $(CC) $(CFLAGS) -Ifunc_inc -c func.c -o func.o

clean:
    rm -f main.o func.o aplikacja
```

Zmiana któregośkolwiek z plików źródłowych (np.: C) wymusza rekompilację tylko tych plików, redukując operacje do tych niezbędnych!

Język C/C++ podstawy budowy programów

```
main.c:
#include <stdio.h>
int main(int argc, char *argv[]){
    printf("Hello world!\n");
    return 0;
}
```

- Jak usprawnić proces kompilacji, cd.
 - Jak wygenerować zależności dla określonego pliku źródłowego
 - zależności do zapisania w pliku Makefile – biblioteki, polecenie:

```
gcc -Wl,-y,printf main.c
```

```
/usr/bin/ld: /lib/x86_64-linux-gnu/libc.so.6: definition of printf
```

- pozwala śledzić symbole i gdzie są implementowane (jeżeli są to funkcje)
- zależności do zapisania w pliku Makefile - pliki nagłówkowe, polecenie:

```
gcc -MM main.c
```

```
main.o: main.c
```

puste zależności bo plik main.c nie ma zależności poza tzw. plikami systemowymi

Język C/C++ podstawy budowy programów

```
main.c:
#include <stdio.h>
int main(int argc, char *argv[]){
    printf("Hello world!\n");
    return 0;
}
```

■ Jak usprawnić proces kompilacji, cd.

■ Jak wygenerować zależności dla określonego pliku źródłowego

■ zależności bardziej szczegółowe - pliki nagłówkowe, polecenie:

```
gcc -M main.c      lub      gcc -M -MG main.c
```

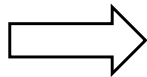
```
main.o: main.c /usr/include/stdc-predef.h /usr/include/stdio.h \
/usr/include/x86_64-linux-gnu/bits/libc-header-start.h \
/usr/include/features.h /usr/include/x86_64-linux-gnu/sys/cdefs.h \
/usr/include/x86_64-linux-gnu/bits/wordsize.h \
/usr/include/x86_64-linux-gnu/bits/long-double.h \
/usr/include/x86_64-linux-gnu/gnu/stubs.h \
/usr/include/x86_64-linux-gnu/gnu/stubs-64.h \
/usr/lib/gcc/x86_64-linux-gnu/8/include/stddef.h \
/usr/lib/gcc/x86_64-linux-gnu/8/include/stdarg.h \
/usr/include/x86_64-linux-gnu/bits/types.h \
/usr/include/x86_64-linux-gnu/bits/typesizes.h \
/usr/include/x86_64-linux-gnu/bits/types/__fpos_t.h \
/usr/include/x86_64-linux-gnu/bits/types/__mbstate_t.h \
/usr/include/x86_64-linux-gnu/bits/types/__fpos64_t.h \
/usr/include/x86_64-linux-gnu/bits/types/__FILE.h \
/usr/include/x86_64-linux-gnu/bits/types/FILE.h \
/usr/include/x86_64-linux-gnu/bits/types/struct_FILE.h \
/usr/include/x86_64-linux-gnu/bits/stdio_lim.h \
...
```

Język C/C++ podstawy budowy programów

■ Łączenie kodu C z fragmentami w assemblerze

- Potrzebne?
- Klasyczne podejście
 - Łączenie plików w C z plikami tworzonymi w ASM
 - Zaawansowane podejście
 - Podobne do łączenia kodu w C z binarnymi bibliotekami
 - Realizowane w fazie linkowania kodu
 - Wymaga znajomości tzw. konwencji przekazywania argumentów i zwracania wyników
 - Pomocne jest użycie polecenia **gcc -S -fno-dwarf2-cfi-asm** aby zobaczyć jak wygląda prolog i epilog funkcji

```
int func(int x){  
    return x+1;  
}
```



```
.text  
.globl _func  
.def    _func; .scl 2; .type 32; .endef  
  
_func:  
LFB0:  
    pushl    %ebp  
LCFI0:  
    movl     %esp, %ebp  
LCFI1:  
    movl     8(%ebp), %eax  
  
    addl     $1, %eax  
  
    popl     %ebp  
LCFI2:  
    ret
```

Nagłówek pliku ASM

Prolog funkcji

Ciało funkcji

Epilog funkcji

Język C/C++ podstawy budowy programów

■ Łączenie kodu C z fragmentami w assemblerze

- Wstawki - gdy fragment, np.: jedna funkcja ma być zapisana w ASM
 - Realizowane w fazie pre-processing kodu

```
#define BIT(n) PORTD=clrClkAndData; \  
asm __volatile__ ( \  
    "sbrc %2," #n \  
    "sbi 18,3" \  
    "sbi 18,5" \  
    "sbic 16,2" \  
    "ori %0,1<<" #n \  
    : "=d" (spiIn) : "0" (spiIn), \  
    "r" (spiOut))
```

```
uint8_t spi(uint8_t spiOut) {  
    uint8_t spiIn = 0;  
    uint8_t clrClkAndData;  
        BIT(7);  
        BIT(6);  
        ...  
        BIT(0);  
    return spiIn;  
}
```

Po pre-procesingu, kompilacji i linkowaniu:

```
//BIT(7)  
ldi        r30, 0x32  
ldi        r31, 0x00  
ldd        r24, Y+1  
st         Z, r24  
ldd        r24, Y+2  
ldd        r25, Y+3  
sbrc       r25, 7  
sbi        0x12, 3      ;0x12=18  
sbi        0x12, 5      ;0x12=18  
sbic       0x10, 2      ;0x10=16  
ori        r24, 0x80    ;0x80=1<<7  
std        Y+2, r24
```

```
//BIT(6)  
ldi        r30, 0x32  
ldi        r31, 0x00  
ldd        r24, Y+1  
st         Z, r24  
ldd        r24, Y+2  
ldd        r25, Y+3  
sbrc       r25, 6  
sbi        0x12, 3  
sbi        0x12, 5  
sbic       0x10, 2  
ori        r24, 0x40    ;0x40=1<<6  
std        Y+2, r24
```

Typy danych języka C – typy proste

(deklaracje, budowa i zasięg)

Typy danych języka C – typy proste

■ Wielkości typów danych

■ char (int8_t)

■ 8 bitów

■ 1-bit znaku

■ 7-bitów wartości

■ min: -128

■ max: 127

■ unsigned char (uint8_t)

■ 8-bitów

■ 8-bitów wartości

■ min: 0

■ max: 255

Typy danych języka C – typy proste

■ Wielkości typów danych

■ short (int16_t)

■ 16 bitów

■ 1-bit znaku

■ 15-bitów wartości

■ min: -32 768

■ max: 32 767

■ unsigned short (uint16_t)

■ 16-bitów

■ 16-bitów wartości

■ min: 0

■ max: 65 535

Typy danych języka C – typy proste

■ Wielkości typów danych

■ long (int32_t)

■ 32-bitów

■ 1-bit znaku

■ 31-bitów wartości

■ min: -2 147 483 648

■ max: 2 147 483 647

■ unsigned long (uint32_t)

■ 32-bitów

■ 32-bitów wartości

■ min: 0

■ max: 4 294 967 296

Typy danych języka C – typy proste

■ Wielkości typów danych

■ long long (int64_t)

■ 64-bitów

■ 1-bit znaku

■ 63-bitów wartości

■ min: -9 223 372 036 854 775 808

■ max: 9 223 372 036 854 775 807

■ unsigned long long (uint64_t)

■ 64-bitów

■ 64-bitów wartości

■ min: 0

■ max: 18 446 744 073 709 551 616

Typy danych języka C – typy proste

■ Wielkości typów danych (IEEE 754)

■ float (32 bitów)

- 1-bit znaku (S)
- 8-bitów wykładnika (E)
- 23-bitów mantysy (M)
- min: $\sim \pm 10^{-38}$
- max: $\sim \pm 10^{38}$

■ double (64 bitów)

- 1-bit znaku (S)
- 11-bitów wykładnika (E)
- 52-bitów mantysy (M)
- min: $\sim \pm 10^{-308}$
- max: $\sim \pm 10^{308}$

Jak wyliczać wartość

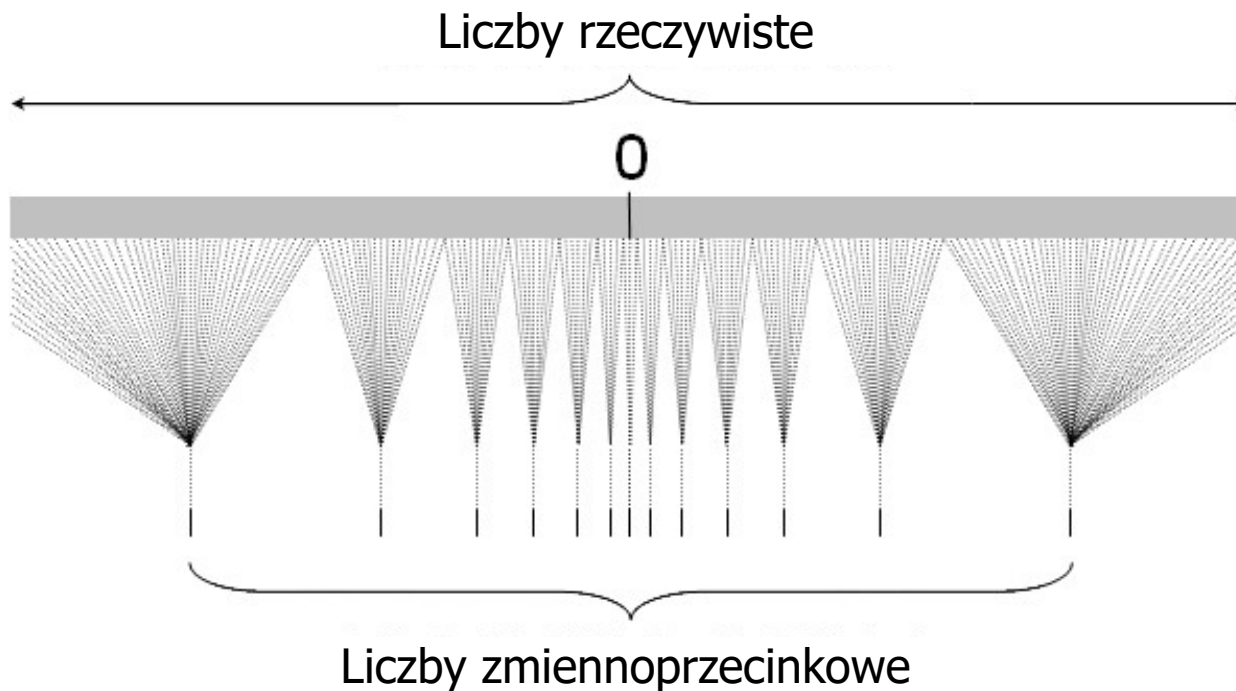
$$x = S * M * 2^E$$

Wartości szczególne (np.: float):

- Nieskończoność („quiet NaN”)
 - $S=0, E=0xFF \Rightarrow -\infty$
 - $S=1, E=0xFF \Rightarrow +\infty$
- lub
 - $E=0xFE \Rightarrow$ jako nie reprezentowalna wartość („signaling NaN”)

Typy danych języka C – typy proste

- Problem obliczeń z użyciem liczb zmiennoprzecinkowych
 - Liczba zmiennoprzecinkowa reprezentuje wiele liczb rzeczywistych
 - Problem brak przechodniości: $F_1 = f(R_1)$ ale $R_2 = f(F_1)$ gdzie $|R_1 - R_2| = d$, $d = f(F_1)$
 - konsekwencja utrata dokładności (np.: przy dodawaniu małych wartości do dużych)
 - rozwiązanie: dodaj wpierw małe wartości, a potem ich sumę do dużych wartości



Zależność między
liczbami rzeczywistymi
a
zmiennoprzecinkowymi

Typy danych języka C – typy proste

■ Wielkości typów danych - problemy

■ typ: int

■ Implementacja zależna od kompilatora

- jego architektury domyślnego traktowania tego typu
- użytych opcji wywołania procesu kompilacji
- kompilator może zatem potraktować „int” jako zmienną o liczbie bitów: 8,16,32,64

■ Zaleca się stosowanie typów o ustalonej liczbie bitów, za pomocą definicji nowych typów, np.:

```
typedef unsigned char uint8_t;  
typedef unsigned short uint16_t;  
typedef unsigned long uint32_t;
```

Typy danych języka C – typy proste

■ Kolejność bitów

■ Bajt (1B)

- Tu wszystkie systemy potrafią się porozumiewać bezbłędnie!

■ Słowo (2B), podwójne słowo (4B), ...

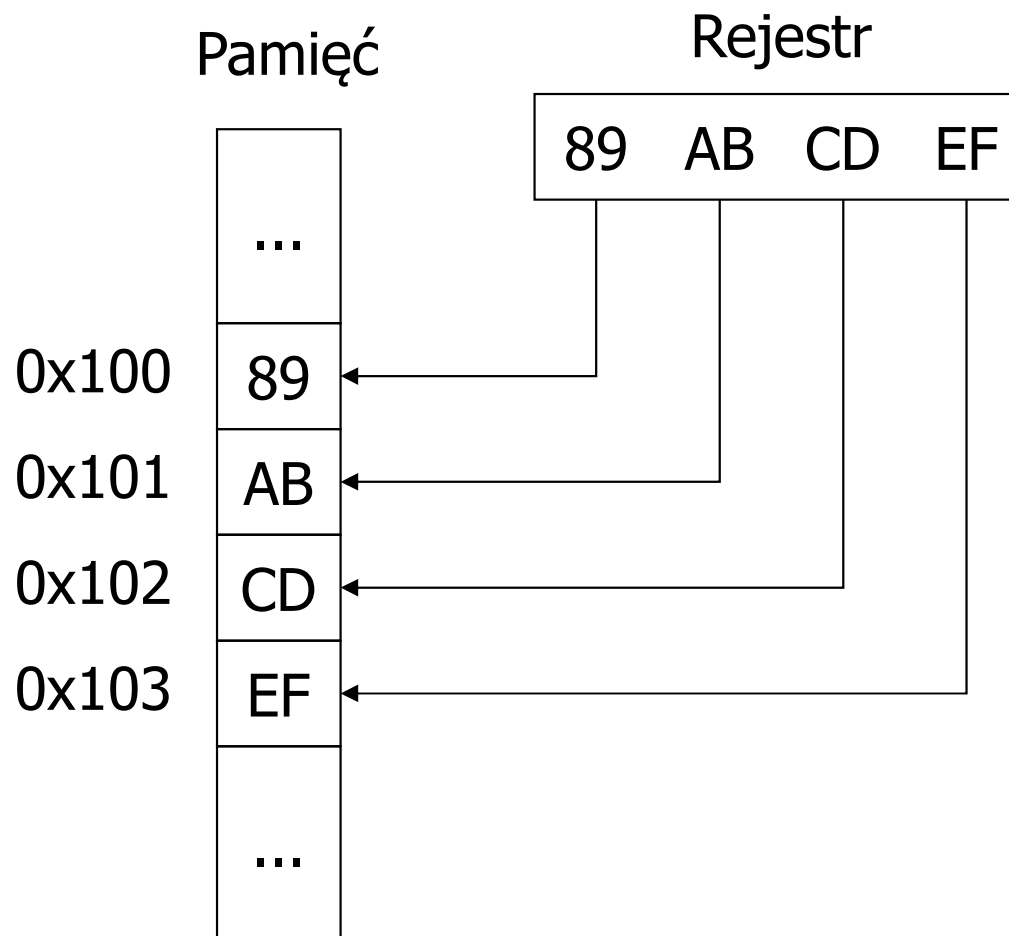
- Tu systemy muszą „rozmawiać” w jednej wybranej notacji:
 - grubo-końcówkowej (big endian)
 - cienko-końcówkowej (little endian)

■ Gdzie występuje problem?

- Operacje zwykłe: *lokalno* - *lokalne*
 - Nie
- Operacje dyskowe: *lokalno* - *”potencjalnie zdalne”* (np.: pendrive)
 - Tak
- Operacje sieciowe: *zdalne*
 - Tak

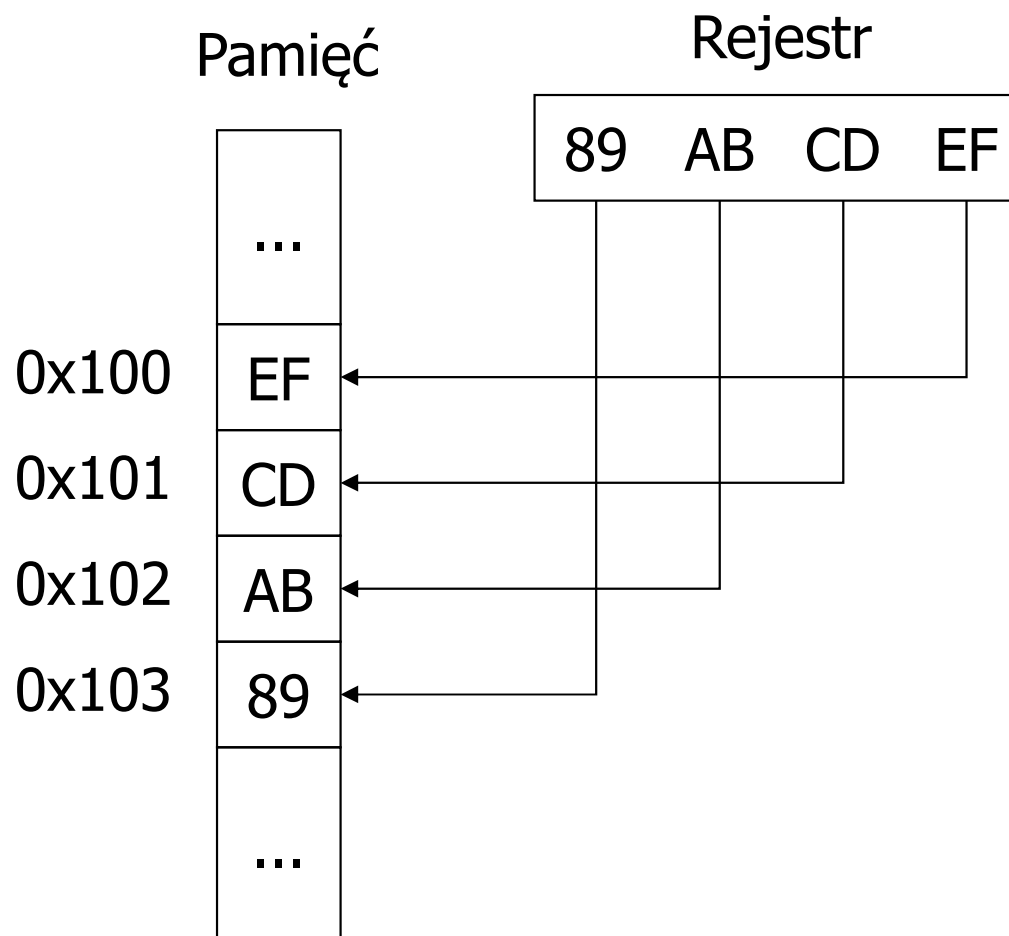
Typy danych języka C – typy proste

- Kolejność bitów, cd .
 - Grubo-kończówkowa (big endian)



Typy danych języka C – typy proste

- Kolejność bitów, cd .
 - Cienko-końcówkowa (little endian)



Typy danych języka C – typy proste

■ Różnica między wartością a znakiem/znakami

- Wartość 65 to 'A', jak zatem pokazać zawartość rejestru R13 równą 65?
- Lub wartość 1234 to w pamięci zapis 0x04D2
- Aby pokazać te wartości (VAL) niezbędna jest konwersja wartości na postać tekstową zgodną z notacją DEC / HEX / OCT / BIN
 - konwersja VAL <-> HEX, VAL <-> OCT, VAL <-> BIN - trywialna
 - konwersja VAL <-> DEC - nieco bardziej skomplikowana

■ Zalety i wady posługiwania się różnymi systemami liczbowymi

- BIN: szybko widać które bity mają jaką wartość, postać rozwlekła
- HEX: postać zwężła, przy pewnej wprawie widać które bity mają jaką wartość, szybka konwersja do OCT/BIN
- OCT: rzadziej stosowana (np.: opis praw dostępu do plików w Linux)
- DEC: postać naturalna człowiekowi, konwersje na inne postaci mocno utrudnione

Typy danych języka C – typy proste

- Różnica między wartością a znakiem/znakami, cd.
 - Jak jednak zamienić liczbę typu „uint32_t” na jej reprezentację tekstową zakładając podstawę 16 – czyli stosując „kod szesnastkowy”?

```
V=0x12345678; //liczba w zmiennej typu uint32_t to „wyświetlenia”
M=0xF0000000; //maska dla najstarszego „nibble” w uint32_t
S=8;          //maks. liczba cyfr w reprezentacji szesnastkowej
Pętla:
    T=(V & M) >> (S*4);
    putch(val2hex((uint8_t)T));
    Jeżeli S<0 zakończ pętlę
    S=S-1;
    M=M>>4; //przestawiamy maskę o 4 bity w prawo
```

- Gdzie

- `val2hex(uint8_t)` – zamienia argument o wartościach od 0 do 15 na liczbę tzw. HEX (0->'0', 1->'1', ..., 9->'9', 10->'A', 11->'B', ..., 15->'F')
- `putch(char)` – wypisuje na „konsolę” dany znak

Typy danych języka C – typy proste

■ Łańcuchy tekstowe

■ Model „PASCAL”

- $\langle \text{liczba znaków} \rangle \langle \text{znak}_1 \rangle \langle \text{znak}_2 \rangle \dots \langle \text{znak}_{\text{liczba znaków}-1} \rangle$

- Zalety: Przechowywane mogą być dowolne znaki

- Wady: Łańcuch ma skończoną długość (wersje o długości 255 lub 65535 znaków), Brak zgodności między wersjami (!)

■ Model „C”

- $\langle \text{znak}_1 \rangle \langle \text{znak}_2 \rangle \dots \langle \text{znak}_N \rangle \langle \text{znak}_{\text{specjalny}} \rangle$

- Zalety: Długość łańcucha niemal nieograniczona (ogranicza wielkość pamięci)

- Wady: Wśród znaków nie można przechowywać znaku specjalnego

■ Inne modele?

Typy danych języka C – typy proste

■ Wartość liczby - NKB

- Dla $N+1$ bitowej liczby A
- Gdzie:
 - $A = (a_N, a_{N-1}, \dots, a_1, a_0)_{\text{NKB}}$
 - a_x = wartość bitu x
- Wartość takiej liczby to

$$A = (-1)^{a_N} * \sum_{i=0}^{N-1} a_i * 2^i$$

- Np.:
 - 0 101 -> wartość: $-1^0 * (2^2 + 2^0) = 1 * (4 + 1) = 5$
 - 1 101 -> wartość: $-1^1 * (2^2 + 2^0) = -1 * (4 + 1) = -5$
- *Wada: podwójne kodowanie wartości 0 (zero)*

Typy danych języka C – typy proste

■ Wartość liczby - U2

- Dla $N+1$ bitowej liczby A
- Gdzie:
 - $A = (a_N, a_{N-1}, \dots, a_1, a_0)_{\text{NKB}}$
 - a_x = wartość bitu x
- Wartość takiej liczby to

$$A = (-a_N) * 2^N + \sum_{i=0}^{N-1} a_i * 2^i$$

- Np.:
 - 0 101 -> wartość: $-0*2^3 + 2^2 + 2^0 = 4 + 1 = 5$
 - 1 101 -> wartość: $-1*2^3 + 2^2 + 2^0 = -8 + 4 + 1 = -3$
- Tu „0” staje się wartością dodatnią ale jest kodowane tylko raz!

Typy danych języka C – typy proste

■ Flaga O (nadmiar) a znaczenie w programie

- Oznacza że wynik operacji nie „zmieścił się” w reprezentowalnej postaci
- O nazywane też
 - V
 - OV
- Definicja
 - Gdy bity znaku obu argumentów są identyczne a znak wyniku przeciwny do nich, wtedy $O=1$, a w przeciwnym razie $O=0$

■ Dla operacji ADC Rd, Rr (zgodnie z: AVR Instruction Set)

- $O = Rd_7 \bullet Rr_7 \bullet !R_7 + !Rd_7 \bullet !Rr_7 \bullet R_7$

Gdzie:

- \bullet - operacja AND
- Rd_7 i Rr_7 - najstarsze bity argumentów
- R_7 - najstarszy bit wyniku

Typy danych języka C – typy proste

■ Flaga *O* (nadmiar) a znaczenie w programie, cd.

■ Przykład dla operacji na danych 7 bitowych ze znakiem:

$$\blacksquare (-53) + (74) = (21) \qquad [0xCB + 0x4A = 0x15]$$

- Mamy przeniesienia z sumowań na pozycjach 6 i 7 więc $O=0$ oraz mimo że $C=1$ (ignorowane) wynik OK.

$$\blacksquare (-53) + (-74) = (-127) \qquad [0xCB + 0xB6 = 0x81]$$

- Mamy przeniesienia z sumowań na pozycjach 6 i 7 więc $O=0$ oraz mimo że $C=1$ (ignorowane) wynik OK.

$$\blacksquare (+54) + (74) = (128) \qquad [0x36 + 0x4A = 0x80]$$

- Mamy przeniesienie z pozycji 6 ale brak przeniesienia z pozycji 7 oraz znak wyniku jest różny od znaków składników daje $O=1$ czyli błędny wynik
- lub zgodnie z dokumentacją dla AVR: $Rd_7=0$ i $Rr_7=0$ a $R_7=1$ stąd $O = 1$
- Wynik tak otrzymany jest błędny bo nie mieści się na zadanej liczbie bitów!

Typy danych języka C – złożone typy danych

(deklaracje, budowa i zasięg)

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

■ Tablice – deklaracja, inicjalizacja

`<typ elementów> nazwa_tablicy [liczba elementów] ...`

■ Deklaracja bez ustalenia wartości - np.:

```
int tablica_liczb[100];  
unsigned char bufor[32];  
double liczby_math[1024];
```

■ Inicjacja tablicy (faza deklaracji) – wielkość określa liczba podanych wartości

```
int tablica_liczb[]={10, 20, 33, 102};
```

■ Inicjacja tablicy (faza deklaracji) – podejście mieszane, podane tylko pierwsze cztery wartości

```
int tablica_liczb[100]={10, 20, 33, 102};
```

■ Inicjacja tablicy (faza używania)

■ deklaracja: offsety od 0 do 99 – całość ma 100 pozycji

```
int tablica_liczb[100];
```

■ użycie: wstawienie liczby na ostatnią pozycję(!)

```
tablica_liczb[99]=1234;
```


Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

■ Tablice - używanie

■ Podejście I

```
#define MAX_INT_ARRAY_SIZE    100  
int tablica_liczb[MAX_INT_ARRAY_SIZE];  
for(i=0; i<MAX_INT_ARRAY_SIZE; i++)  
    tablica_liczb[i]=...  
...
```

■ Podejście II

```
typedef int melement;  
melement tablica_liczb[100];  
for(i=0; i<sizeof(tablica_liczb)/sizeof(melement); i++) ...
```

■ Operator „sizeof” – zwraca wynik typu size_t (!)

- sizeof(unsigned char) = sizeof(char) = 1 (zawsze o ile nie przeciążone!)

```
char mstring[32] = "witam";
```

- Operator sizeof(mstring) zwróci 32

- Funkcja strlen(mstring) zwróci 5

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

■ Tablica - ułożenie w pamięci

■ Nazwa tablicy to alias dla adresu w pamięci gdzie ją umieszczono(!)

- Odwołania realizowane są poprzez wskazanie adresu w pamięci a na niskim poziomie typowo przez instrukcje - x86: MOV, PUSH/POP, Risc-V: LD/ST

■ „Język C” nie sprawdza offsetów (!!!)

- Takie zachowanie staje się źródłem wielu problemów, niejednokrotnie źródłem nadużyć

■ Ciekawostka mało znana

- Odwołanie `tablica_liczb[20]` jest tożsame z `20[tablica_liczb]` (!)

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

■ A łańcuchy tekstowe?

- Pomagają np.: w komunikacji z użytkownikiem

- Deklaracja z inicjacją i proste użycie niskopoziomowe – jedno z wielu rozwiązań:

```
char *mtekst = "Ala ma kota";  
int i;  
for (i=0; ; i++){  
    if (mtekst[i]=='\0')  
        break;  
    my_putchar(mtekst[i]);  
}
```

Tu niejawnie wstawiony zostanie znak o kodzie '\0'

Przechodzimy po kolejnych lokacjach, aż do napotkania znaku '\0'

Tu komunikujemy się ze sprzętem (np.: port szeregowy, konsola ekranowa) i wypisujemy jeden znak za pomocą tego sprzętu

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

■ Struktury – zestaw wielu zmiennych

```
struct _nazwa_definicji_struktury {  
    definicje_pol_skladowych;  
} nazwa_deklaracji_struktury;
```

■ Odwołania do pól – przykład

```
struct _zestaw_rejestrow{  
    unsigned char A;  
    ...  
} zestaw_rejestrow;  
...  
zestaw_rejestrow.A = 1;
```

■ Nazwa struktury w przeciwieństwie do tablic nie jest aliasem adresu w pamięci

■ Adres to:

```
struct _zestaw_rejestrow *p = & zestaw_rejestrow;
```

■ A odwołanie do pól to:

```
p -> A = 1;
```

Typy danych języka C – złożone typy danych

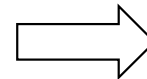
■ Przypomnienie – typy zaawansowane w języku C

- Uwaga! nowoczesne kompilatory języka C dla zwiększenia szybkości programu wynikowego rozmieszczają pola domyślnie w adresach będących wielokrotnością słowa maszyny
 - np.: kompilacja dla X86 gdzie słowo maszyny ma 32bity

■ Przykład

```
struct _zestaw_rejestrow{  
    unsigned char A;  
    unsigned short FLAGS;  
    unsigned long PC  
} zestaw_rejestrow;
```

Możliwe rozmieszczenie pól w pamięci



A	-	-	-
FLAGS		-	-
PC			

■ Istnieje sposób aby upakować pola

```
struct _nazwa_definicji_struktury {  
    ...  
} nazwa_deklaracji_struktury __attribute__((packed)) ;
```

■ Przydatne gdy należy zachować zgodność bitową pól

- Wymiana takich struktur z innymi maszynami
 - np.: nagłówki plików z „treścią binarną”: EXE, AVI, MP3, DOCX, ...

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

■ Struktury z polami binarnymi

```
struct _nazwa_definicji_struktury {  
    definicje_pol_skladowych : liczba_bitow;  
} nazwa_deklaracji_struktury;
```

■ Zastosowania - przykłady

■ Interakcja ze sprzętem

■ Obsługa nietypowych typów danych

- specyficzna kompresja danych – przykład struktury o wielkości 2B (zamiast 3B)

```
struct _czas{  
    char godzina: 4; //zał. w aplikacji nie ma znaczenia czy am/pm  
    char minuta: 6;  
    char sekunda: 6;  
} czas __attribute__((packed));
```

- uwaga! kod wynikowy staje się większy o obsługę pól bitowych

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

■ Unie

- Agregacja wielu typów w jednej strukturze (Polymorphic Data Structures)

```
union nazwa_unii {  
    definicje_pol_skladowych;  
}
```

- Unia zajmuje w pamięci wielkość największego pola
- Zastosowanie - z ręczne żonglowanie między typami danych
 - Przykład – użycie pól w „opcode” RISC-V

```
union risc_v_opcode{  
    uint32_t raw;  
    struct _u_type{  
        uint32_t imm:    20;  
        uint32_t rd:      5;  
        uint32_t opcode:  7;  
    } u_type;  
    ...  
};  
...
```

```
...  
uint32_t ExU2(uint32_t val){...}  
...  
union risc_v_opcode opcode;  
...  
opcode.raw=MEM[PC];  
if(opcode.u_type.opcode==AUIPC){  
    REG[opcode.u_type.rd]=PC+ExU2(opcode.u_type.imm);  
    ...  
}  
...
```

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

■ Tablice i tablice struktur/unii

- Typy takie ułatwiają organizację danych w pamięci podczas działania programu
 - Przypomnienie – „język C” nie sprawdza offsetów w odwołaniach do pól tablicy(!!!)
- Załóżmy że mamy w kodzie deklaracje i użycie:

```
#define MAX_STUDENTS 100  
  
int tablica_ocen_lab[MAX_STUDENTS];  
  
...  
  
tablica_ocen_lab[100]=2;    //w tablicy ostatni element ma offset 99(!)
```

- Błędny jest tu użyty offset 100, gdy wynikające z deklaracji offsety to od 0 do 99 włącznie

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

- Tablice i tablice struktur/unii – zagrożenia: odwołania do „obcej” pamięci
 - Wyjście poza obszar przydzielonej pamięci – przypadek statyczny
 - Przypadek prosty do wykrycia

```
#define MAX_ELEMENTS 100
...
typedef struct{
    unsigned int lab, egzamin;
} student_ocena;
...
student_ocena tablica_ocen[MAX_ELEMENTS];
...
tablica_ocen[102].lab=2;
...
```

- Tzw. „ręczna” analiza kodu wykryje ten błąd – widać że 102 jest z poza zakresu 0...99

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

- Tablice i tablice struktur/unii – zagrożenia, odwołania do „obcej” pamięci, cd.
- Wyjście poza obszar przydzielonej pamięci – przypadek statyczny
 - Przypadek nieco bardziej skomplikowany – wymaga dokładnej analizy kodu

```
#define MAX_ELEMENTS 100
typedef struct{
    unsigned int lab, egzamin;
} student_ocena;
student_ocena tablica_ocen[MAX_ELEMENTS];
#define REFERENCE_VALUE_OFFSET    (50)
#define DEFAULT_VALUE_OFFSET      (REFERENCE_VALUE_OFFSET*2+1)
tablica_ocen[DEFAULT_VALUE_OFFSET].lab=3;
```

Ile równe jest to makro?



- Przydatne jest tu „przejście” przez preprocesor - wywołując *gcc -E main.c* otrzymamy:

```
typedef struct{
    unsigned int lab, egzamin;
} student_ocena;
student_ocena tablica_ocen[100];
tablica_ocen[101].lab=3;          //tu już widać problem (!)
```

Typy danych języka C – złożone typy danych

■ Przypomnienie – typy zaawansowane w języku C

- Tablice i tablice struktur/unii – zagrożenia, odwołania do „obcej” pamięci, cd.
 - Przypadek dynamiczny – wychodzimy poza obszar przydzielonej pamięci podczas działania aplikacji (często efekt powstaje tylko przy pewnych danych wejściowych)

```
int correction=90; //zmienna globalna ustalana w wielu miejscach, zależnie od
                  //danych wejściowych

int func(int i){    //funkcja pomocnicza
    return i<correction ? correction*2 : i; //jakaś obróbka danych
}

int pos=0;
...
pos--;              //tej operacji można w swoim kodzie „nie zauważyć”
tablica_ocen[pos].lab=2; //efekt: niejawnie wyszliśmy poza tablicę
...
pos=20;             //pozornie wskazanie jest ok
tablica_ocen[func(pos)].lab=2; //efekt: niejawnie także wyszliśmy poza tablicę
```

■ Pomoc

- czytanie ostrzeżenia generowane przez kompilator
- programy do automatycznej analizy kodu np.: Valgrind [Valgrind.org]

Typy danych języka C – złożone typy danych

■ Rola stosu w systemie komputerowym

- Przypomnienie – m.in. miejsce przechowywania zmiennych lokalnych, argumentów wywołania i adresów powrotu
- Typowa organizacja pamięci („C memory layout”)

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

```
int globalna_zmienna;
```

```
int globalna_zmienna_zainicjowana=0;
```

```
void func(int parametr){
```

```
    int zmienna_lokalna; //widziana  
                        //tylko w func()
```

```
    ...
```

```
}
```

```
int main(){
```

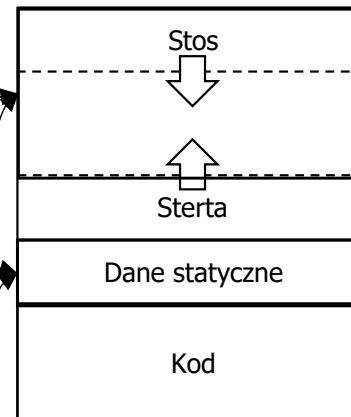
```
    int zmienna_lokalna;
```

```
    static int zmienna_statyczna;
```

```
    ...
```

```
    return 0;
```

```
}
```



Zmienne lokalne, adresy powrotu, argumenty wywołania

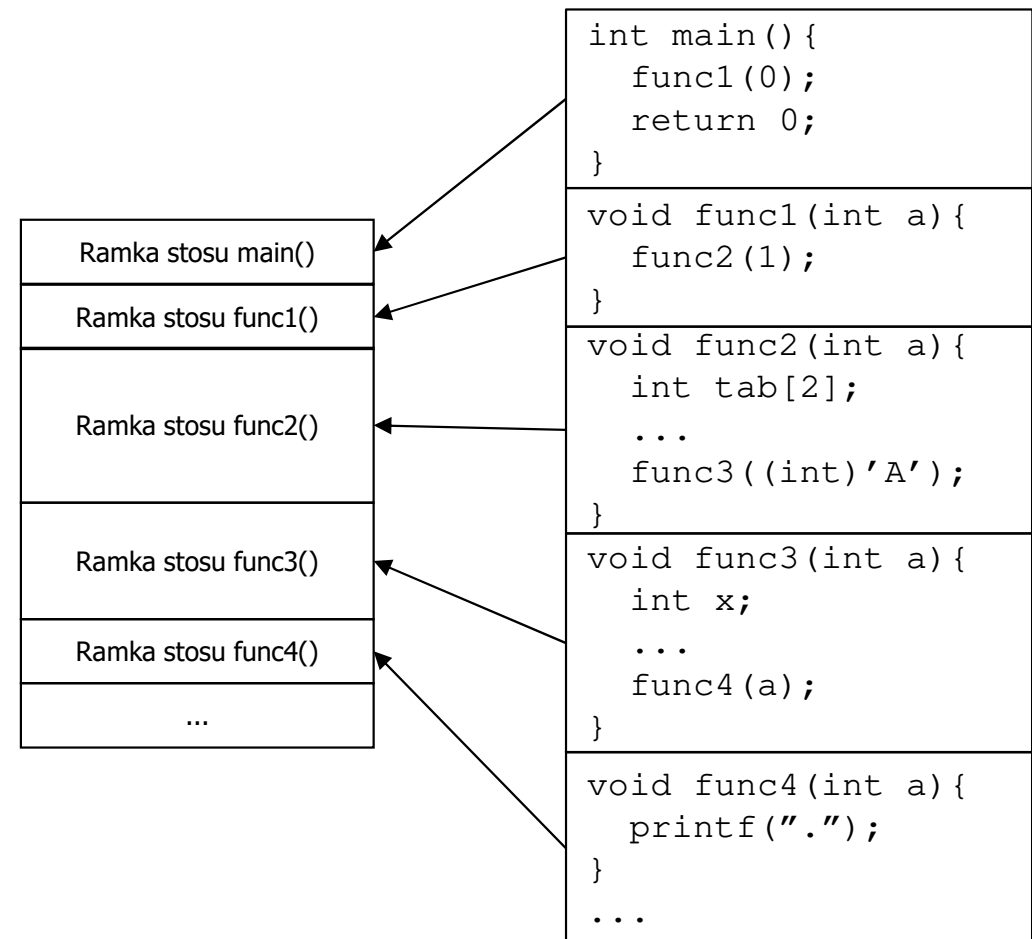
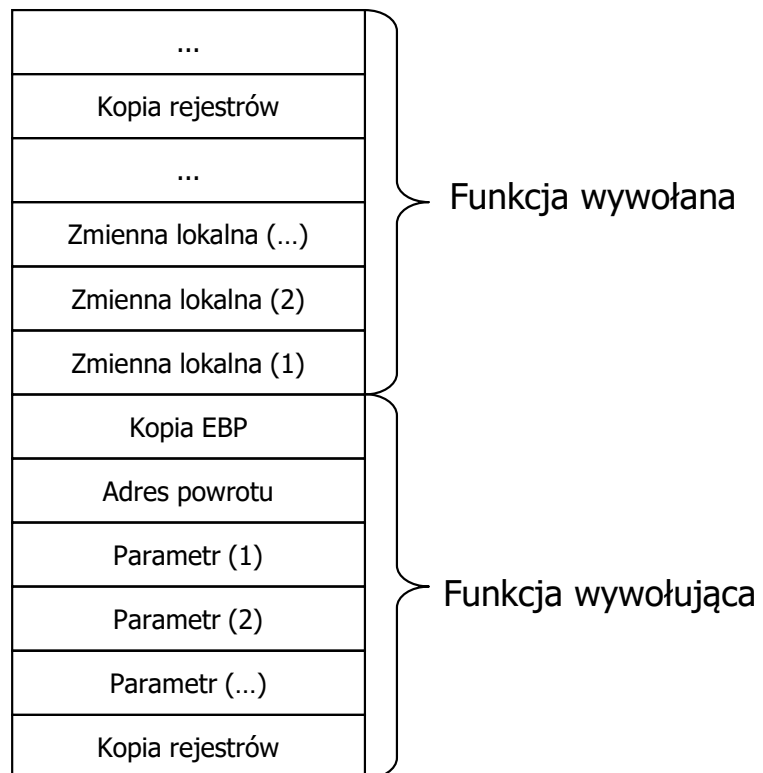
Obszar dynamicznie alokowanych obiektów/zmiennych za pomocą:
w C i C++ funkcji: malloc/free (realloc, calloc, ...),
w C++ (i tylko w tam) operatorów: new/delete

Zmienne globalne, statyczne – ich rozmiar nie ulega zmianie, są w pamięci przez cały czas działania kodu aplikacji/systemu

Typy danych języka C – złożone typy danych

■ Rola stosu w systemie komputerowym

- Przypomnienie – m.in. miejsce przechowywania zmiennych lokalnych, argumentów wywołania i adresów powrotu, cd.
- Kompilatory języka C stosują tzw. „ramkę stosu” (stack frame)
 - EBP – tzw. „Base Pointer” w X86



Typy danych języka C – złożone typy danych

■ Przypomnienie – zasięg zmiennych w języku C

- Zmienne lokalne – pomagają kompilatorowi w zarządzaniu pamięcią
 - Zmienna „znika” po wyjściu z funkcji (!)
 - jej miejsce w pamięci może być użyte do innych celów
 - Liczenie, że wskazany obszar po wyjściu z funkcji będzie zawierał tę samą wartość jest błędem

Kod błędny – czasami może działać (!)

```
int *func(void){
    int x=5;
    return &x; //pozornie poprawne
    //przekazanie adresu zmiennej x -
    //kompilator z reguły o tym ostrzeże
}
int main(){
    int *z,x,y;
    z=func();
    x=*z;                //lub x=z[0];
    printf("%d\n", x); //tu otrzymamy 5
    y=*z;
    printf("%d\n", y); //wynik będzie nie
                       //znany(!)

    return 0;
}
```

Kod poprawny

```
void func(int *z){
    int x=5;
    *z=x; //lub z[0]=x;
}
int main(){
    int z,x,y;
    func(&z);
    x=z;
    printf("%d\n", x); //tu otrzymamy 5
    y=z;
    printf("%d\n", y); //tu także otrzymamy 5

    return 0;
}
```

Typy danych języka C – kolejność operacji

- W języku C kolejność operacji (ang. Operator Precedence) w wynikowym jest ściśle określona

Wyższy priorytet

Niższy priorytet

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	other
<pre>a = b a += b a -= b a *= b a /= b a %= b a &= b a = b a ^= b a <=< b a >=> b</pre>	<pre>++a --a a++ a--</pre>	<pre>+a -a a + b a - b a * b a / b a % b ~a a & b a b a ^ b a << b a >> b</pre>	<pre>!a a && b a b</pre>	<pre>a == b a != b a < b a > b a <= b a >= b</pre>	<pre>a[b] *a &a a->b a.b</pre>	<pre>a(...) a, b (type) a ?: sizeof _Alignof (since C11)</pre>

Masz wątpliwości to zamiast pisać:

```
c = a > b ? a++ : b + 1
```

napisz:

```
c = ((a < b) ? (a++) : (b + 1))
```

lub (gdy miałeś co innego na myśli):

```
c = ((a < b) ? (a++) : b) + 1
```

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal(c99)	
2	++ --	Prefix increment and decrement[note 1]	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of[note 2]	
	_Alignof	Alignment requirement(c11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	?:	Ternary conditional[note 3]	Right-to-left
14[note 4]	=	Simple assignment	
	+= -=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
	<<= >>=	Assignment by bitwise left shift and right shift	
15	&= ^= =	Assignment by bitwise AND, XOR, and OR	
	,	Comma	Left-to-right

Typy danych języka C – rzutowanie

■ Rzutowanie (ang. casting)

- Czyli zamiana zmiennej jednego typu na inny typ (czasami jest to operacja chwilowa, wykonywana na potrzeby określonej operacji)

(nowy typ) zmienna

- Np.:

```
float a = 3.14;
```

```
int b = (int)a + 1; //zmienna 'b' będzie miała wartość 4 (!)
```

- Po co – gdy chcemy obserwować zmienną o określonym typie jako zmienną o innym typie
 - Pamiętajmy, że typowy procesor nie potrafi dodawać zmiennych o różnych typach

Typy danych języka C – rzutowanie

■ Rzutowanie (ang. casting), cd.

- Jak kompilator traktuje w kodzie źródłowym np.: dodawanie, $A=B+C$
 - Jeżeli B i C są tego samego typu zadanie proste – kompilator wybiera instrukcję procesora (add, ...) lub funkcję biblioteczną (_add_uint32, ...) obsługującą operacje dodawania dla tego typu
 - Jeżeli typy się nie zgadzają to kompilator dokonuje promocji argumentu do właściwego typu, np.:

`char -> short, short -> long, long -> long long, long long -> float, float -> double, ...`

- Na przykład gdy zmienne zdefiniowano: `short B; char c`, kompilator wykona niejawne rzutowanie:

`B + (short) C`

- A co kompilator zrobi z wynikiem? Gdy zdefiniowano A jako `float`, wtedy wynik musi być innego typu niż B i C, kompilator tutaj także dokona niejawnego rzutowania co doprowadzi do wyrażenia:

`A = (float) (B + (short) C);`

- lub (zależnie od strategii optymalizacji):

`A = (float) B + (float) C;`

Typy danych języka C – rzutowanie

■ Enkapsulacja przez rzutowanie

```
...
uint16_t my_udp_port=...;
...
uint8_t ProcessUDP(uint8_t *rxb, uint16_t n, uint8_t *txb){
    uint8_t ret=0;
    struct EtherFrame *ef=(struct EtherFrame*)&rxb[0];
    struct IpFrame *ipf=(struct IpFrame*)&((ef->data)[0]);
    struct UdpFrame *udpf=(struct udp*)&((ipf->data)[0]);
    if(ntohs(udpf->rport)==my_udp_port){
        uint16_t len=MainApp((void*)udpf->data, n, txb);
        ...
    }
    return ret;
}
uint16_t MainAPP(void *payload, uint16_t n, uint8_t *txb){
    uint16_t ret;
    ...
    return ret;
}
```

```
struct EtherFrame{
    uint8_t  DestHWADD[6];
    uint8_t  SourHWADD[6];
    uint16_t typeHW;
    uint8_t  data[1];
};
```

```
struct IpFrame{
    uint8_t  verlen;
    uint8_t  tos;
    uint16_t len;
    uint16_t id;
    uint16_t frag_off; //+flags
    uint8_t  ttl;
    uint8_t  proto;
    uint16_t h_checksum;
    uint8_t  src[4];
    uint8_t  dst[4];
    uint8_t  data[1];
};
```

```
struct UdpFrame{
    uint16_t lport;
    uint16_t rport;
    uint16_t len;
    uint16_t cksum;
    uint8_t  data[1];
};
```

Typy danych języka C – złożone typy danych

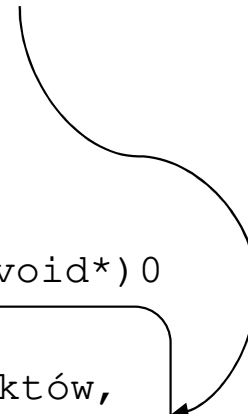
■ Rola stery w systemie komputerowym

■ Przypomnienie – używanie pamięci dynamicznie alokowanej

■ Alokacja

```
#define LICZBA_STUDENTOW 32
typedef struct {
    int lab, egzamin;
} ocena_t;                //definiujemy obiekt
ocena_t *oceny;           //tu tylko deklarujemy wskaźnik
...
oceny=(ocena_t *)malloc(LICZBA_STUDENTOW * sizeof(ocena_t));
if(oceny == NULL){        //NULL - specjalna wartość typowo: (void*)0
    printf("Brak pamięci! (F:%s, L:%d)\n", __FILE__, __LINE__);
    ...                  //zwolnienie innych dynamicznie utworzonych obiektów,
    ...                  //zamknięcie plików, zamknięcie gniazd sieciowych, ...
    exit(1);
}
...                      //operacje na obiekcie wskazanym przez zmienną 'oceny'
free(oceny);              //zwalniamy z pamięci obiekt 'oceny'
```

Tu może pojawić się problem jak przetestować ten fragment!



■ Zwalnianie pamięci dynamicznie zaalokowanej to obowiązek programisty (!)

- Mimo że w C nie ma automatycznego Garbage Collector'a, to wiele systemów operacyjnych usuwając proces usuwa także jego struktury dynamicznie zaalokowane – ale błędem jest zakładać że to wystarczy

Typy danych języka C – złożone typy danych

■ Przepelnienie stosu – niebezpieczeństwa

- Stos ma skończoną pojemność i może jego wielkość nie być chroniona
- Przykład błędnego kodu

```
void func(void){
    uint32_t buffer[20];
    ...
    strcpy((char*)buf, „NAPIS_DLUZSZY_NIZ_20_ZNAKOW”); //tu zostanie
                                                         //zamazany adres powrotu - co powodować
                                                         //będzie np.: zawieszenie programu,
                                                         //niestabilność
    ...
    buf[1003]=BAD_CODE_ADDRESS; //typowe nadużycie - tu adres
                                //powrotu podmieniany jest na skok do
                                //złośliwego kodu
    ...
}
```

int main(void){

...

func();

...

}

Gdy ten adres będzie precyzyjnie ustalony - tak by trafić w obszar pamięci stosu gdzie składowany jest adres powrotu z funkcji main() - może ułatwić wykonanie niedozwolonych operacji po zakończeniu wykonywania tego kodu, przejmując prawa tego kodu

Adres taki może zależeć w danej aplikacji, od wersji jej kodu, od wersji użytego kompilatora i od wersji systemu operacyjnego – generalnie trudne do ustalenia ale możliwe (!)

Istnieją w systemach operacyjnych funkcję randomizacji rozmieszczania w pamięci określonych funkcji

Typy danych języka C – złożone typy danych

■ Funkcje „niebezpieczne” w C

- Biblioteka LIBC zawiera funkcje obsługi ciągów tekstowych

- Wersje niebezpieczne:

```
size_t strlen(const char *string);  
int strcmp(const char *s1, const char *s2);  
int strcpy(char *dst, const char *src);  
int sprintf(char *str, const char *format, ... );  
char *strcat(char *dst, const char *src);  
...
```

- Wersje bezpieczne:

```
size_t strlen(const char *s, size_t maxlen);  
int strncmp(const char *s1, const char *s2, size_t maxlen);  
int strncpy(char *dst, const char *src, size_t maxlen);  
int snprintf(char *str, size_t maxlen, const char *format, ... );  
char *strncat(char *dst, const char *src, size_t maxlen);  
...
```

- Wersje bez swojego bezpiecznego odpowiednika: atol(), atof(), atoi(), ...

Dziękujemy za uwagę!