Programmering, grundkurs, 8.0 hp HI1024, extra tentamen, TEN1, för TIDAA1

Fredagen den 11 mars 2011, 13.15 – 17.15

Tentamen består av två delar, del A och del B. Del A innehåller 10 kryssfrågor på olika teman inom *C*-programmering. Varje fråga är uppbyggd på ett påstående. Påståendet är antingen *helt korrekt* och är då alltså SANT, eller så finns *minst ett allvarligt fel* i påståendet och då är alltså påståendet FALSKT. Man kryssar det man tror på och om man kryssar rätt får man 1 poäng. Kryssar man fel får man –1 poäng. Det är alltså riskabelt att chansa. Man har alltid alternativet att kryssa AVSTÅR som då ger 0 poäng. Del A kan inte ge mindre än 0 poäng totalt. (Om man har mer fel är rätt får man alltså inga negativa poäng.) Del B innehåller 13 traditionella frågor som ni ska svara skriftligt på. Totalt antal poäng är 32. Gränsen för E är 14 och gränsen för Fx är 12. KTHs allämnna regler för examination gäller. (Gränserna för E och Fx är sänkta med 2 poäng.)

Del A.

Påstående 1. Ett *C*-programs "källkod" och "maskinkod" är två olika namn för samma sak.

[] SANT [] FALSKT [] AVSTÅR

Påstående 2. Man kan skicka heltalsarrayer till funktioner som parametrar genom funktionsprototyperna

`funk (int * arr)` eller `funk (int arr[])`.

Skillnaden är att då man gör deklarationen `int arr[]` är det möjligt för funktionen `funk` att ta reda på antal element i arrayen, det är inte möjligt om man användet deklarationen `int * arr`. Då skickar man lämpligen arrayens längd som en andra parameter.

[] SANT [] FALSKT [] AVSTÅR

Påstående 3. Om en array har 10 platser numreras dessa från 0 till 9. Om man försöker skriva till plats nr 10 (som ju inte finns) så försöker man skriva utanför det minnesområde som hör till arrayen. *C* upptäcker detta och avbryter då programkörningen.

[] SANT [] FALSKT [] AVSTÅR

Påstående 4. En liten forsättning på föregående fråga, ett (dåligt) sätt att hantera ovanstående problem skulle kunna vara att sätta på en modulooperator (`%`) vid varje indexberäkning till arrayen, det vill säga om vi har heltalsarrayen `int arr[10]` så varje gång vi vill komma åt ett element på plats `i` så skriver vi `arr[i%10]`. Modulooperatorn ser till att uttrycket `i%10` alltid håller sig mellan 0 och 9 och garanterar på detta sätt att vi inte försöker skriva utanför arrayens minnesområde. Det fungerar men det är ett dåligt sätt, vi vill ju helst lösa det grundläggande frågan varför vi över huvud taget försöker skriva utanför arrayen och inte sopa det problemet under mattan på detta sätt.

[] SANT [] FALSKT [] AVSTÅR

Påstående 5. Antag att vi har deklarationen

```
struct person {
  char namn[20];
  char adress[30];
  int alder;
};
```

Det är då möljigt för oss att skriva en funktion med prototypen

```
lagg_in (struct person p, char namn[], char adress[], int alder);
```

som lägger in uppgifterna namn, adress och ålder i `p` som tydligen är en struct av den typ som anges ovan. Ett anrop skulle kunna se ut så här:

```
lagg_in (person1, "Kalle", "Skogstigen 3", 42);
```

Variabeln `person1` antas deklarerad som en struct av ovanstående typ.

[] SANT [] FALSKT [] AVSTÅR

Påstående 6. Om vi har två heltal `a` och `b` där `b` är större än `a` och vi vill beräkna hur många procent `a` är av `b` så är denna procentsats rent matematiskt lika med *100 * a / b*. Ett vanligt fel är då att bilda uttrycket `proc = 100 * a / b`. Detta kommer att bli noll eftersom kvoten mellan `a` och `b` är ett heltal som blir 0 (eftersom `b` var större än `a`). För att få *C* att uföra en flyttalsoperation och alltså hantera `a / b` som ett decimaltal, mindre än 1, måste man se till att `proc` deklareras som flyttal, det vill säga satserna

```
float proc;
int a = 10, b = 20;
proc = 100 * a / b;
```

kommer att resultera i att `proc` får värdet 50, som vi vill, för `a` är 10 som är 50% av `b` som är 20. Uttrycket `a / b` kommer här alltså att hanteras som decimaltalet 0.50 (tack vare deklarationen `float proc`).

[] SANT [] FALSKT [] AVSTÅR

Påstående 7. Man kan läsa in ett innehåll i en sträng med `scanf` men då kan man inte läsa mellanslag eller tabulatorsteg, vi kan alltså skriva

```
char namn[20];
scanf("%s",namn);
```

och korrekt läsa in namn som inte innehåller mellanslag (eller tabulatorsteg).

[] SANT [] FALSKT [] AVSTÅR

Påstående 8. Förprocessordirektiv kan hjälpa oss att definera värden i ett *C*-program som ska gälla överallt i programmet. Vi kan då få en mer elegant programkod om detta kombineras med `switch`-satser.Vi kan skriva

```
#define MONDAY 0
#define TUESDAY 1
#define WEDNESDAY 2
#define THURSDAY 3
#define FRIDAY 4
#define WEEKEND 5
```

och i programmet ha koden

```
switch(day)
{
  case MONDAY: case TUESDAY: case WEDNESDAY: case THURSDAY: case FRIDAY:
    printf("Work on, no weekend yet...\n");
    break;
  case WEEKEND:
    printf("Weekend! No worries!\n");
    break;
}
```

(Här förutsätts `day` vara deklarerad på lämpligt sätt och programkoden förutsätts förekomma i ett lämpligt sammanhang i någon funktion, kanske `main()`. `#define`-direktiven finns längst upp i programmet.)

[] SANT [] FALSKT [] AVSTÅR

Påstående 9. Olika variabler kan ha olika typer och därmed lagra olika sorters data. Olika datatyper har också olika utrymmeskrav, en `char` tar 1 byte, en `short int` tar 2 och en `int` tar normalt 4. En `float` tar normalt 4 och en `double` tar 8. Rent tekniskt kan man dock hävda att allt som hanteras egentligen är ettor och nollor, begreppet "datatyp" existerar således för att ge en tolkning av den bitföljd (ettor och nollor) som lagras. Tolkningen innebär att beräkningar och presentationer kan utföras på ett, för människan konsistent sätt, men datorsystemet själv har mycket dåliga begrepp om vad det arbetar med, det är bland annat därför som *C* inte alltid varnar då man begår ett typfel.

[] SANT [] FALSKT [] AVSTÅR

Påstående 10. En switch-sats är uppbyggd av en kontroll av ett heltalsuttryck, därefter följer ett block av `case`-delar och eventuellt en `default`-del. Mycket ofta skiljs `case`-delarna år av `break`-satser, när man kör break är det klart och exekveringen lämnar det aktuella blocket, här alltså hela `switch`-satsen. När `switch`-satsen dock är en del av en funktion behövs ibland inte `break` om vi efter körning av en viss `case`-del har etablerat ett returvärde, då kan bara `return` köras vilket fullbordar exekveringen av `switch`-satsen även utan `break`. Ett exempel nedan:

```
float day_salary (int day)
{
  switch(day)
  {
  case MONDAY: case TUESDAY: case WEDNESDAY: case THURSDAY: return 100.0;
  case FRIDAY: return 150.0;
  case WEEKEND: return 200.0;
  }
}
```

Vi tänker oss här en funktion som anger en viss lön, lönen är 100.0 på vardagar utom fredag då den är 150 och på helgen är den 200. Här slipper vi `break`-satser som sagt, eftersom `return` fullbordar exekveringen av `switch`-satsen på ett korrekt sätt.

[] SANT [] FALSKT [] AVSTÅR

Del B. (Nödvändiga `#include`-direktiv är ofta utelämnade, men ska anses finnas där ändå.)

Uppgift 11. (2p) Studera nedanstående program:

```
main()
{
  int i,j;

  for(i=1;i<=5;i++)
  {
    for(j=1;j<=5;j++)
      if (i%j!=0 && j%i!=0) printf("(%d,%d) ", i, j );
    printf("\n");
  }
}
```

Ange den utskrift som uppkommer då man kör det.

Uppgift 12. (2p) Studera nedanstående program:

```
main()
{
  char str1[] = "MCE R";
  char str2[] = "YKTBA";
  char str[30];

  int i=0;

  for(i=0;i<10;i++)
    if(i%2==0)str[i]=str1[i/2];else str[i]=str2[i/2];

  str[10]='\0';

  printf("%s\n", str);
}
```

Ange den utskrift som uppkommer då man kör det.

Uppgift 13. (2p) Studera nedanstående program:

```
main()
{
  int i, a[] = {3, 2, 4, 1, 0};
  char s[] = "BLANDAD";

  for(i=0;i<=4;i++)
    printf("%d ", s[b[i]] );
  printf("\n");
}
```

Ange den utskrift som uppkommer då man kör det. (Ledning: se upp med typen på `s[b[i]]`, vad innebär `%d` här?)

Uppgift 14. (1p) Man önskar utskriften

```
2 5 10 17 26
```

(alltså de första fem jämna kvadraterna + 1) av programsnutten

```
i=...;
while(i<=...)
{
  printf("%d ", ...);
  i=...;
}
```

Fyll i det som behövs för att det ska fungera som tänkt.

Uppgift 15. (2p) Studera nedanstående program:

```
int f1(int a, int *b)
{
  a=10-a;
  *b++;
  printf("F1 %d %d\n", a, *b);
  return a;
}

f2(int j, int i)
{
  int z;
  j++; i--;
  z=f1(i+j, &i);
  printf("F2 %d %d %d\n", i, j, z);
}

main()
{
  int i=10,j=20;
  f2(i,j);
  printf("MAIN %d %d\n", i, j);
}
```

Ange den utskrift som uppkommer då man kör det.

Uppgift 16. (3p) Studera nedanstående program, det är tänkt att användas för beräkning av ersättningsresistanser vid serie- och parallellkopplingar av två resistorer vars resistanser betecknas r1 och r2. Ersättningsresistansen blir r1+r2 i seriefallet och r1*r2/(r1+r2) i parallellfallet.

```
double get_positive() {
  double tmp = 1.0;
  while(tmp<0.0)
  {
    printf("Ett positivt tal: ");
    scanf("%lf",&tmp);
  }
  return tmp;
}

main() {
  double r, r1, r2; int val;

  printf("Ange resistans 1: "); r1=get_positive();
```

```
  printf("Ange resistans 2: "); r2=get_positive();

  while (val!=1 && val!=2)
  {
    printf("Serie eller parallellkoppling\n");
    printf("1) Serie eller 2) Parallel: ");
    scanf("%d",&val);
  }

  if(val==1)
    printf("r: %d\n", r1+r2);
  else
    printf("r: %d\n", r1*r2/r1+r2);
}
```

Programmet har fem felaktiga rader. Rätta dessa fel. Felen är av logisk karaktär så det genereras inga kompileringsfel, dock genereras två varningar. Ange även anledningen till dessa två varningar.

Uppgift 17. (2p.) I *Del A* förekom flera falska påståenden. Välj två av dessa falska påståenden och förklara vad som egentligen gäller i sammanhanget, det vill säga uttryck vad som skulle vara sant i sammanhanget och varför det är på det viset.

Uppgift 18. (3p) Betrakta nedanstående program, det slumpar fram fem strängar och vill sortera dem i bokstavsordning:

```
random_str(char str[]) {
  int pos;
  for(pos=0;pos<9;pos++)
    str[pos]='a'+rand()%25;
  str[pos]='\0';
}

main()
{
  char str[5][10];
  char tmp[10];
  int i, j, min=0;
  srand(time(0));
  for(i=0;i<5;i++)random_str(str[i]);
  for(i=0;i<5;i++)printf("test: %s\n", str[i]);

  //Sortering:

  for(i=0;i<5;i++) {
    min=i;
    for(j=i+1;j<5;j++)
      if(strcmp(str[min],str[j])>0)min=j;
    if(min!=i)
    {
      strcpy(tmp,str[i]);
      strcpy(str[i],str[min]);
      strcpy(str[j],tmp);
    }
  }

  printf("\n");
  for(i=0;i<5;i++)printf("test: %s\n", str[i]);

}
```

Det finns dock ett fel någonstans i sorteringen, finn felet och rätta det!

Uppgift 19. (2p) I *Del A* förekom flera falska påståenden. Välj ytterligare två av dessa falska påståenden och förklara vad som egentligen gäller i sammanhanget, det vill säga uttryck vad som skulle vara sant i sammanhanget och varför det är på det viset. (Du måste förstås välja andra påståenden än du redan behandlat i uppgift 17.)

Uppgift 20. (3p) Arean av en cirkel med radien 1 med centrum i origo i planet är som bekant lika med Pi som är ungefär lika med 3.1416... Detta brukar kallas *enhetscirkeln*. I olika sammanhang vill man kunna beräkna ett bättre värde på Pi. Vi skulle kunna beräkna ett värde på Pi genom att slumpa punkter (x,y) i kvadraten med sidan 2 som ligger parallellt med koordinataxlarna i xy-planet med centrum i origo, denna kvadrat omsluter precis enhetscirkeln. Arean på kvadraten är 4 och arean på enhetscirkeln är, som sagt Pi. Då skulle vi kunna få ett närmevärde på Pi genom att beräkna 4*antal punkter i cirkeln/total antal punkter. Det talet bör närma sig Pi ju fler punkter vi slumpar ut. Nedanstående program är tänkt att göra precis detta. Det fungerar dock inte som det är tänkt. Dert finns tre fel i programmet. Finn felen och rätta dem!

```
main()
{
  int i, antal_i;

  double x, y, pi;

  srand(time(0));

  for(i=0;i<100000;i++)
  {
    x = ((double)(rand()%32766))/16383-1.0;
    y = ((double)(rand()%32766))/16383-1.0;
    if(x*x+y*y<=2.0)antal_i++;
  }

  pi = (4.0*antal_i)/1000000;

  printf("Pi: %lf.\n", pi);

}
```