Temat: Funkcje- argumenty funkcji, przekazywanie argumentu przez wskaźnik.

Argumenty funkcji - nazwy argumentów

Jak już dobrze wiesz, funkcje mogą przyjmować argumenty lub nie. W poprzedniej lekcji przedstawiłem Ci różne funkcje - zarówno te, które przyjmowały argumenty, jak i takie, które argumentów nie przyjmowały.

Lista argumentów funkcji składa się z nazwy typu oraz nazwy argumentu funkcji. Powstaje pytanie - jak powinny być nazywane argumenty - dowolnie, czy może tak jak nazwy zmiennych, które zostały do funkcji przekazane?

Okazuje się, że nazwy argumentów funkcji mogą nazywać się dowolnie pod warunkiem, że w danym zasięgu nazwy argumentów nie będą się powtarzać.

W poniższym programie zostały zadeklarowane 2 funkcje, o nazwach **funkcja1** i **funkcja2**. Jak możesz zauważyć, obie funkcje wywoływane są z argumentami **liczba1** i **liczba2**. Jednak w deklaracji funkcji argumenty te są nazwane różnie. W funkcji **funkcja1** argumenty te są nazwane inaczej niż nazwy argumentów wywołania funkcji w funkcji main. Natomiast w funkcji **funkcja2** argumenty te są nazwane dokładnie tak samo jak nazwy argumentów wywołania funkcji w funkcji main. Jak widać, w obu przypadkach funkcje działają poprawnie, co oznacza że w deklaracji funkcji argumenty mogą być nazwane albo tak samo jak nazwy argumentów podczas wywołania funkcji lub też zupełnie inaczej.

```
#include <iostream>

using namespace std;

void funkcja1(int, int);
double funkcja2(int, int);



int main()
{
  int liczba1=5,liczba2=8;

  funkcja1(liczba1,liczba2);
  cout <<"Suma liczb to "<<funkcja2(liczba1,liczba2);


  cout <<endl<<"Nacisnij ENTER aby zakonczyc"<<endl;
  getchar();
  return 0;
}

void funkcja1(int a, int b)
{
 cout <<a<<' '<<b<<endl;
}

double funkcja2(int liczba1, int liczba2)
{
 return liczba1+liczba2;
}
```

Jak dobrze wiesz, w programie nazwy zmiennych nie mogą powtarzać się w ramach jednego zasięgu. Jak to jest w przypadku funkcji?

Każda funkcja wyznacza swój własny zasięg. Zmienne oraz etykiety deklarowane wewnątrz funkcji nie są znane po zakończeniu wywołania danej funkcji.

Oznacza to zatem, że również argumenty kilku różnych funkcji mogą mieć identyczne nazwy. Gdybyśmy w poprzednim przykładzie chcieli zmienić deklarację funkcji funkcja1 i zamiast:

```
void funkcja1(int a, int b)
{
  cout <<a<<' '<<b<<endl;
}
```

chcielibyśmy napisać

```
void funkcja1(int liczba1, int liczba2)
{
```

```
 cout <<liczba1<<' '<<liczba2<<endl;
}
```

to nie stanowiłoby to żadnego problemu. Co prawda w takiej sytuacji argumenty obu funkcji (funkcja1 i funkcja2) nazywałyby się dokładnie tak samo, ale właśnie z powodu, że każda funkcja wyznacza swój własny zasięg, nie spowodowałoby to żadnego konfliktu.

Problemem natomiast byłaby deklaracja zmiennych wewnątrz funkcji o nazwach takich samych jak nazwy argumentów funkcji. Takie zagadnienie obrazuje poniższy przykład (definicja przykładowej funkcji):

```
void funkcja1(int a, int b)
{
 int a=2, b=8;
 cout <<a<<' '<<b<<endl;
}
```

W takim przypadku kompilator zaprotestuje i kompilacja programu się nie powiedzie - najprawdopodobniej ujrzysz komunikat w stylu **declaration of 'int a' shadows a parameter**. Kompilator protestuje z prostego powodu - w tym samym zakresie są zadeklarowane zmienne o takich samych nazwach i nawet gdyby deklaracja zmiennych wewnątrz funkcji była możliwa, niemożliwe byłoby skorzystanie z argumentów funkcji - a skoro możliwe byłoby doprowadzenie do tak absurdalnej sytuacji, nie jest to po prostu możliwe.

Funkcje a zmienne globalne

Skoro funkcje mogą przyjmować argumenty, powstaje pytanie czy w ogóle powinny mieć dostęp do zmiennych globalnych? W zasadzie każda wartość zmiennej może zostać przekazana do funkcji właśnie za pomocą argumentów. Po co zatem funkcje miałyby korzystać ze zmiennych globalnych?

Okazuje się, że funkcje mogą korzystać ze zmiennych globalnych, jednak nie należy nadużywać tej możliwości. Pamiętaj, że funkcja stanowi jakiś fragment programu i funkcje zostały wprowadzone po to, aby łatwiejsze było również testowanie programu.

Wyobraź sobie, że mamy program, w którym zadeklarowaliśmy 30 zmiennych globalnych i 100 różnych funkcji, przy czym funkcje korzystają ze zmiennych globalnych - zarówno czytają wartość tych zmiennych, jak i je zmieniają. Nagle okazuje się, że jakaś funkcja pobiera wartość zmiennej globalnej, ale ta wartość nie jest prawidłowa. Najprawdopodobniej została ona zmieniona w nieprawidłowy sposób przez którąś z pozostałych 99 funkcji. Co w takiej sytuacji? Musiałbyś sprawdzić po kolei 99 funkcji, aby udało Ci się znaleźć błąd, co wcale mogłoby nie być takie łatwe.

Poniższy program demonstruje wykorzystanie zmiennych globalnych w funkcjach. Każda z funkcji przedstawia nieco inne zagadnienie.

```
#include <iostream>

using namespace std;

const double a = 6;
double pole=0;


double pole1(double);
double pole2(double);
double pole3(double);

void informacja();


int main()
{
 informacja();

 cout <<endl<<"Pole dla bokow 6 i 3 wynosi "<<pole1(3)<<endl;
 informacja();

 cout <<endl<<"Pole dla bokow 6 i 15.5 wynosi "<<pole2(15.5)<<endl;
 informacja();

 cout <<endl<<"Pole dla bokow 6 i 7 wynosi "<<pole3(7)<<endl;
 informacja();

 cout <<endl<<"Nacisnij ENTER aby zakonczyc"<<endl;
 getchar();
 return 0;
```

```
}

double pole1(double b)
{
 pole = a * b;
 return pole;
}

double pole2(double a)
{
 pole = ::a * a;
 return pole;
}

double pole3(double c)
{
 unsigned int a = 1; // ilosc prostokatow
 pole = ::a * c * a;
 return pole;
}


void informacja()
{
  cout <<"Zmienna a ma wartosc "<<a
      <<". Zmienna pole ma wartosc "<<pole<<endl;
}
```

Jak widzisz w programie znajdują się definicje 2 zmiennych globalnych (zmienna **a** i zmienna **pole**), przy czym zmienna **a** jest opatrzona modyfikatorem **const**. Ponadto w programie zostały zadeklarowane 4 funkcje. Zadaniem funkcji **informacje** jest wyświetlanie informacji na temat wartości zmiennych globalnych.

Zasadniczym elementem programu są funkcje **pole1**, **pole2** i **pole3** - wszystkie funkcje mają jedno podstawowe zadanie - liczą powierzchnię prostokąta, którego jeden bok ma długość równą wartości zmiennej globalnej a, natomiast drugi bok ma długość, która zostaje przekazana jako argument wywołania tych funkcji.

Każda z funkcji wykorzystuje **zmienną globalną a** do obliczenia pola prostokąta i każda z tych funkcji obliczoną wartość pola prostokąta zapisuje w **zmiennej globalnej pole**. Każda z funkcji zwraca ponadto wartość obliczeń - w przypadku naszych funkcji wartość obliczeń jest równa oczywiście zmiennej globalnej pole.

Przede wszystkim zauważ, że każda z 3 funkcji obliczających pole, wykorzystuje zmienne globalne bez najmniejszego problemu. **Zmienna a** jest wykorzystywana do obliczenia, natomiast **zmienna pole** jest wykorzystywana do zapisania wyniku obliczeń. Jak zatem widać, funkcje mogą korzystać bez problemu ze zmiennych globalnych, w tym również modyfikować ich wartość.

Funkcja **pole1** dokonuje prostego obliczenia. Ponieważ w funkcji tej nie ma nigdzie definicji zmiennej o nazwie takiej jak zmienna globalna a, funkcja ta może odwołać się bezpośrednio do zmiennej globalnej a i zapis w takim przypadku wygląda dokładnie tak samo jak w przypadku zmiennej lokalnej, czyli zadeklarowanej wewnątrz tej funkcji.

Z kolei funkcja **pole2** posiada zdefiniowany argument o nazwie a, czyli został tutaj wykorzystany mechanizm przesłaniania zmiennych. Oczywiste jest zatem to, że samo podanie nazwy zmiennej spowoduje pobranie wartości zmiennej lokalnej, czyli w tym przypadku argumentu funkcji. Jak jednak widać, skorzystanie ze zmiennej globalnej jest nadal możliwe, ale poprzez wykorzystanie **operatora zasięgu ::** (o tym operatorze była już mowa w lekcji o przesłanianiu zmiennych).

Podobne zagadnienie demonstruje funkcja **pole3**, z tym tylko, że tym razem to nie argument funkcji wykorzystuje mechanizm przesłaniania zmiennych, a zmienna lokalna zdefiniowana wewnątrz funkcji. Jak widzisz, podobnie jak dla funkcji pole2, wykorzystanie zmiennej globalnej jest możliwe dzięki wykorzystaniu operatora zasięgu. Jednocześnie chcę zwrócić Twoją uwagę na to, że oczywiście dla obliczenia pola prostokąta pomnożenie długości boków przez zmienną lokalną a o wartości 1, nie jest do niczego potrzebne - jednak zostało tutaj wykorzystane po prostu do zobrazowania możliwości skorzystania również ze zmiennej lokalnej.

Ostatnią kwestią pozostaje to, czy zademonstrowane w powyższym programie funkcje pole1, pole2 i pole3 powinny w ogóle korzystać ze zmiennych globalnych.

Korzystanie ze stałej **zmiennej globalnej a** można jeszcze zrozumieć - przykładowo w jakiejś fabryce produkowane są blachy zawsze o jednakowej długości i różnej szerokości - w takim przypadku wygodnie jest skorzystać ze zmiennej globalnej, która przechowuje stałą długość. W przeciwnym przypadku, do każdej funkcji należałoby przekazać 2 argumenty, z czego drugi

miałby zawsze wartość 6 - w ten sposób zaoszczędzamy czas programowania i możliwość ewentualnego błędu (przekazanie złej wartości lub złej zmiennej).

Jednak już przypisywanie przez każdą z tych funkcji wyniku obliczeń do **zmiennej globalnej pole** jest mocno dyskusyjne. Oczywiście mogłoby się zdarzyć, że wartość ta jest wykorzystywana przez jeszcze jakieś inne funkcje, jednak w naszym programie funkcje nie wykorzystują tej zmiennej do odczytu, a jedynie do zapisania tam wartości obliczeń. Zatem takie wykorzystanie zmiennej globalnej raczej nie powinno mieć miejsca, bowiem może stać się tylko powodem poważnych błędów w całym programie.

Przekazywanie argumentów do funkcji przez wskaźnik

Mimo że przekazywanie argumentów do funkcji przez referencję rozwiązuje wiele problemów, to jednak sama metoda stwarza jeden zasadniczy problem - jeśli spojrzysz na wywołanie funkcji, do której jest przekazywany argument przez referencję, to zauważysz, że wywołanie to nie różni się niczym od przekazywania argumentu do funkcji przez wartość. W pierwszym momencie możesz pomyśleć, że to bardzo dobrze, bo dzięki temu nie musisz uczyć się dodatkowych zasad i wywołanie takiej funkcji jest bardzo proste.

Jeśli jednak spojrzysz na to zagadnienie z perspektywy bardzo dużego programu, to już wtedy nasuwa się na myśl problem, jaki się może pojawić - wywołując funkcję, do której przekażesz argumenty, nie będziesz tak naprawdę wiedzieć czy może ona zmodyfikować takie argumenty czy nie. Jest to bardzo poważny problem, bowiem analiza takiego programu jest znacznie trudniejsza. Oczywiście jeśli przyjrzysz się definicji (a nawet samej deklaracji) takiej funkcji, to już będziesz wiedzieć, że funkcja przyjmuje argumenty przekazywane poprzez referencję, jednak w przypadku dużego programu składającego się na przykład ze 100 różnych plików takie sprawdzanie deklaracji funkcji jest niepotrzebną stratą czasu.

Wykorzystanie przekazywania argumentów do funkcji przez wskaźnik rozwiązuje ten problem - sposób przekazywania do funkcji w tym przypadku widać zarówno w deklaracji/definicji funkcji, jak również podczas wywołania funkcji. Poniżej znajduje się przykład wykorzystania przekazywania argumentów do funkcji przez wskaźnik:

```
#include <iostream>

using namespace std;

void zmien(double *, unsigned int *);

int main()
{
 double a=5;
 unsigned int b=6;

 cout <<"Zmienna a wynosi "<<a<<". Zmienna b wynosi "<<b<<endl;
 zmien(&a,&b);
 cout <<"Po zmianie: Zmienna a wynosi "<<a<<". Zmienna b wynosi "<<b<<endl;

 cout <<endl<<"Nacisnij ENTER aby zakonczyc"<<endl;
 getchar();
 return 0;
}

void zmien(double *liczba1, unsigned int *liczba2)
{
 *liczba1 += 4;
 *liczba2 = *liczba2 * 3;
}
```

Jak łatwo możesz stwierdzić, również w tym przypadku - podobnie jak dla przekazywania argumentów przez referencję, wartości zmiennych przekazanych do funkcji zachowują wartość zmienioną przez funkcję. Tutaj jednak podobieństwo się kończy.

W definicji i deklaracji funkcji znajduje się znak ***** oznaczający wskaźnik. Natomiast podczas wywołania funkcji podczas przekazywania argumentów, pojawia się znak **&**. Na pierwszy moment mógłbyś pomyśleć, że to referencja, jednak referencja to na pewno nie jest. Znak **&** oznacza bowiem w tej sytuacji operator pobrania adresu zmiennej.

Po raz kolejny, podobnie jak w przypadku referencji, wielokrotnie widziałem problemy w zrozumieniu przez początkujących programistów dlaczego tak wygląda definicja funkcji i dlaczego tak wygląda przekazywanie argumentów do funkcji. Należy postąpić podobnie jak w przypadku referencji.

Przypomnij sobie jak wygląda **schemat inicjalizacji** wskaźnika:

```
typ *nazwaWskaznika = &nazwaZmiennej;
```

Potraktuj teraz poszczególne elementy listy argumentów w definicji funkcji jako lewą stronę powyższego schematu (czyli to co znajduje się po lewej stronie operatora przypisania), a jako prawą stronę powyższego schematu potraktuj miejsce wywołania funkcji. W taki sposób otrzymasz:

```
double *liczba1 = &a;
unsigned int *liczba2 = &b;
```

Jak zatem widzisz, kod odpowiada w 100% schematowi inicjalizacji wskaźnika, który Ci przypomniałem. Jeśli teraz skorzystasz z odwrotnego rozumowania tzn. znasz schemat inicjalizacji wskaźnika (lub w końcu zapamiętasz) i będziesz chciał zadeklarować, a następnie używać funkcji, która ma argumenty przekazywane przez wskaźnik, to nie będziesz już nigdy mieć wątpliwości jak powinna wyglądać definicja i wywołanie tego typu funkcji.

Zadanie:

Na dzisiejszej lekcji przeanalizuj podane w treści programy.

Następnie do zadań z poprzedniej lekcji użyj metod opisanych w dzisiejszej lekcji.

Wyślij na adres: marek@zstio-elektronika.pl programy w których metody z dzisiejszej lekcji.

Termin niedziela godz:12:00.