Funkcje

- Rola funkcji jest intuicyjnie jasna. Jeżeli jakiś fragment programu wykonywany jest wielokrotnie, być może z różnymi danymi, wydzielamy go jako funkcję.
- Funkcję musimy nazwać, i działają tutaj te same zasady tworzenia nazw, co w przypadku zmiennych.
- W przypadku funkcji z reguły rozdzielamy definicję i deklarację. Pamiętamy, że w przypadku zmiennych z reguły używamy tylko definicji.
- W przypadku funkcji regułą jest, że deklaracja funkcji umieszczona jest na samym początku pliku, przed pierwszym użyciem funkcji. Natomiast definicja, która zawiera całą implementację, umieszczona jest gdzie indziej, często w osobnym pliku.

Funkcje

- Funkcja przyjmuje pewną ilość *argumentów*, i zwraca jedną wartość. Deklaracja zawiera nazwę funkcji, oraz typy przyjmowanych argumentów. Typowa deklaracja:

```
int czescCalkowita( double x );
```

- Funkcja może nie przyjmować żadnego argumentu. Wtedy nawias jest pusty. Ale zawsze musi być nawias.

```
int wypiszMenu();
```

- Dopuszczalna, ale przestarzała (pozostałość z języka C), jest w takim przypadku także składnia

```
int wypiszMenu( void );
```

Funkcje

- Funkcja może nie zwracać żadnej wartości. Typem zwracanym jest wtedy `void`

```
void wczytajDane();
```

- Deklaracja funkcji zawiera podstawowe informacje, które pozwalają kompilatorowi sprawdzić, czy funkcja jest w programie prawidłowo wywoływana. W zasadzie nawet nazwy argumentów nie są potrzebne, tylko typy. Całkowicie prawidłowa jest więc taka deklaracja

```
int pomnoz( double, double );
```

  Jednak z reguły nazwy też zamieszczamy. Tak jest wygodniej. Z reguły deklarację po prostu kopiujemy z pierwszej linijki definicji funkcji i wklejamy w odpowiednie miejsce. Pamiętajmy o średniku.

Funkcje

- Program może więc wyglądać następująco:

```
#include <iostream>

using namespace std;

int czescCalkowita( double x );

double x = 2.5;

int main()
{
    double x;
    cout << "Podaj x: ";
    cin >> x;
    cout << "Część całkowita x to: "
         << czescCalkowita( x ) << endl;
    cout << "Część całkowita globalnej x to: "
         << czescCalkowita( ::x ) << endl;
  return 0;
}

int czescCalkowita( double x )
{
    return x;
}
```

Funkcje

- Zauważmy, że deklaracja jest identyczna z nagłówkiem definicji, z wyjątkiem średnika.
- Jeżeli funkcja zwraca wartość, to musi zawierać instrukcję `return`. To działa tak samo, jak w funkcji głównej `main()`. Jeżeli funkcja nie zwraca żadnej wartości, to instrukcję `return` można pominąć. Takich instrukcji `return` może być wiele. W momencie napotkania tej instrukcji wykonywanie funkcji jest zakończone, a odpowiednia wartość (która oczywiście może być różna dla różnych `return`) jest podstawiana w miejscu wywołania funkcji.
- Zauważmy, że funkcja `czescCalkowita()` powinna zwrócić liczbę typu `int`, a zwraca liczbę typu `double`. Ma tu miejsce niejawne rzutowanie typu, o którym mówiliśmy już omawiając zmienne. W zasadzie takich niejawnych rzutowań powinno się unikać, ponieważ utrudniają one czytanie i zrozumienie kodu.

Funkcje

- W programie nazwa `x` pojawia się kilkakrotnie. Nie ma z tym żadnego problemu, ponieważ ich zakresy ważności są rozłączne. Te `x` które występują w deklaracji funkcji `czescCalkowita()`, lub w jej deklaracji to tak zwane parametry formalne. Ich zasięg jest ograniczony do bloku w którym występują, czyli definicji lub deklaracji. Zmienna `x` które występuje w funkcji `main()` to zmienna lokalna dla funkcji `main()`, która jest podawana przez użytkownika i przekazana jako argument do funkcji `czescCalkowita()`przy pierwszym wywołaniu.
- Zmienna `x` która jest zdefiniowana poza obiema funkcjami to zmienna globalna, która jest dostępna wszędzie wewnątrz pliku. Jej nazwa jest *przesłonięta* przez ewentualne zmienne lokalne o tej samej nazwie. Nie ma więc konfliktu. Zmienną globalną można odsłonić używając operatora `::`. Przy ponownym wywołaniu funkcji `czescCalkowita()` została jej przekazana zmienna globalna.

Funkcje

- Zmienne występujące w deklaracji czy definicji funkcji nazywamy parametrami formalnymi funkcji. Natomiast zmienne, które zostają przekazane funkcji w momencie wywołania często nazywa się argumentami funkcji.
- Zmienne zadeklarowane w deklaracji czy nagłówku definicji funkcji to tak zwane zmienne automatyczne. Nie są one tworzone w pamięci w momencie uruchomienia programu. Zostają utworzone, w momencie wywołania funkcji, w obszarze pamięci zwanym stosem.
- W dominującej architekturze systemów procesorowych (tak zwanej architekturze von Neumanna) cała pamięć dostępna zorganizowana jest w jednowymiarową listę kolejnych komórek. Program, zmienne globalne (i literały) oraz wszystkie funkcje umieszczone są na samym dole tej listy w momencie uruchomienia programu. Natomiast na samym końcu listy znajduje się tak zwany stos, którego wielkość stale się zmienia.

Funkcje

- Jak sama nazwa wskazuje, dostęp do stosu jest przez jego wierzchołek. Jest to dostęp typu *first in last out*. Procesor może utworzyć jakieś zmienne na stosie, lub je ze stosu usunąć. O ile wszystkie obiekty umieszczone „na dole” pamięci mają swoje raz na zawsze ustalone adresy, o tyle zmienne utworzone na stosie mają różne przypadkowe adresy, związane ze stanem stosu, przy którym zostały utworzone.
- W momencie wywołania funkcji na stosie zostają automatycznie (stąd ich nazwa) utworzone zmienne odpowiadające parametrom formalnym funkcji. Zmiennym tym zostają *przekazane* jakieś wartości. To przekazanie może się odbyć „przez wartość”. Tak to się odbyło w powyższym przykładzie. Zmiennym na stosie przypisano wartości występujące w wywołaniu. Funkcja operuje na zmiennych na stosie, a oryginalne zmienne (tak jak w przykładzie zmienna globalna `x` pozostają niezmienione.

Funkcje

- Rozważmy następujący przykład

```
#include <iostream>

using namespace std;

int potroj( int x );

int main()
{
    int x;
    cout << "Podaj x: ";
    cin >> x;
    cout << endl << "Potrojone x to: " << potroj( x ) << endl;
    cout << "Jeszcze raz potrojone: "
         << potroj( potroj( x ) ) << endl;
    cout << "Wyjściowe x to: " << x << endl;
    return 0;
}

int potroj( int x )
{
    x *=3;
    return x;
}
```

Funkcje

- Otrzymujemy przykładowy wynik:

```
Podaj x: 5

Potrojone x to: 15
Jeszcze raz potrojone: 45
Wyjściowe x to: 5
```

- Zauważmy, że wyjściowa zmienna globalna nie zmieniła się, pomimo kilku wywołań funkcji `potroj()` z argumentem `x`. Argument został przekazany funkcji przez wartość, czyli funkcja operuje na kopii zmiennej. `x` występujące w definicji funkcji, które w trakcie jej wywołania zostaje pomnożone przez 3, to kopia oryginalnej zmiennej globalnej. Kopia ta zostaje usunięta z pamięci po zakończeniu wykonywania funkcji instrukcją `return`. Zostają wtedy usunięte ze stosu wszystkie zmienne automatyczne

Funkcje

- Zwróćmy uwagę na zagnieżdżenie wywołania funkcji `potroj()`. Nie ma z tym problemu. Kiedy program wywołuje funkcję (tą na zewnątrz), tworzy zmienną automatyczną na stosie, i kiedy przechodzi do przypisania jej wartości, funkcja wywoływana jest ponownie. Program tworzy kolejną zmienną automatyczną na stosie, i przypisuje jej wartość `x`. Wartość tej ostatniej zmiennej automatycznej zostaje przez funkcję pomnożona przez 3, po czym wykonanie funkcji kończy się, potrojona wartość jest pobierana ze stosu, stos zostaje zmniejszony, a otrzymana właśnie wartość przypisana poprzedniej zmiennej automatycznej, która teraz czeka na szczycie stosu. I tak dalej.
- Za moment zobaczymy przykład funkcji, która wywołuje sama siebie. Nie ma w tym niczego dziwnego, kolejne wywołania po prostu powiększają stos.

Funkcje

- Jest też inny mechanizm przekazywania argumentu funkcji, tak zwane przekazanie „przez referencję”. Argument jest tak przekazywany, jeżeli parametr formalny występuje ze znakiem &. Zmodyfikujmy nasz przykład:

```
...

int potroj( int &x );

int main()
{

   ...

}

int potroj( int &x )
{

   ...

}
```

- Jedyna zmiana to znak & przed parametrem funkcji.

Funkcje

- W języku C++ ten znak oznacza adres. Wielokrotnie będziemy się z nim spotykali w przyszłości. Samo wywołanie funkcji jest takie jak poprzednio. (Musimy jednak usunąć zagnieżdżone wywołanie. To się właśnie wiąże z faktem, że zmienne na stosie nie mają ustalonego adresu.)

```
Podaj x: 5

Potrojone x to: 15
Wyściowe x to: 15
```

- Tak jak mówiliśmy, tym razem funkcja operowała nie na kopii zmiennej globalnej, ale na samej tej zmiennej. Do zmiennej automatycznej na stosie został przypisany adres (referencja) zmiennej globalnej `x`, a nie jej wartość.

Funkcje

- W C++ funkcja zwraca tylko jedną wartość. Jeżeli chcielibyśmy, żeby zwracała więcej, to możemy wykorzystać do tego właśnie wywołanie przez referencję. Wyobraźmy sobie, że chcemy mieć funkcję, która przyjmie promień koła jako argument, a zwróci jego pole i obwód

```
const double pi = 3.14;

void poleObwod(double promien, double &outPole, double &outObwod);

int main()
{
  double promien, pole, obwod;
  cin >> promien;
  poleObwod( promien, pole, obwod );
  cout << pole << endl << obwod;
}

void poleObwod(double promien, double &outPole, double &outObwod)
{
  outPole = pi * promien * promien;
  outObwod = 2 * pi * promien;
}
```

Funkcje

- W ten sposób pomimo, że formalnie funkcja nic nie zwraca (typ zwracany `void`), to faktycznie zwraca nam pole i obwód koła.
- Tak użyte parametry funkcji nazywa się czasem parametrami wyjściowymi. Podkreśliliśmy to w deklaracji funkcji używając opisowych nazw. To może ułatwić osobie czytającej zorientowanie się, co tak naprawdę funkcja robi.
- Takie wykorzystanie przekazywania argumentów przez referencję nie jest jednak rekomendowane. Problemem jest to, że w miejscu wywołania funkcji nie ma żadnego ostrzeżenia, że argument przekazujemy przez referencję. Jeżeli rzeczywiście tak chcielibyśmy zdefiniować funkcję, powinniśmy użyć jeszcze jednego sposobu przekazania argumentu, tak zwanego przekazania przez wskaźnik

Funkcje

- O wskaźnikach będziemy mówili w przyszłości, w skrócie chodzi o to, że wywołując funkcję musimy podać jej argument typu właśnie wskaźnikowego, i jest od razu jasne, jaki jest charakter przekazania.
- To, że w C++ funkcja może zwracać tylko jedną wartość nie jest ograniczeniem, a raczej filozofią C++. Ta pojedyncza zwracana wartość może być rozbudowaną strukturą zawierającą wiele składników. Z reguły właśnie tak jest.

Funkcje

- Jest jeszcze jeden aspekt tej sprawy. Dobry obyczaj pisania programów stanowi, że każda funkcja robi tylko jedną rzecz. Funkcje są proste, ale jest ich wiele. Rozbudowane funkcje rozbijamy na proste składniki, które łatwo jest zrozumieć, i łatwo znaleźć błędy. Taka prosta funkcja powinna przyjmować tylko minimalną ilość niezbędnych parametrów, których nie rusza, i zwracać jeden konkretny „produkt", czymkolwiek miałby on być. Prosta logiczna konstrukcja.
- Idea przekazywania argumentu przez referencję jest taka, że czasem argument jest wielki. Na przykład funkcja musi coś zrobić z wielką tablicą, powiedzmy posortować. Przepisywanie wielkiego obiektu na stos będzie długo trwało, i stos będzie niepotrzebnie duży. To jest sytuacja, gdzie przekazywanie przez referencję ma sens. jest.

Funkcje

- Jeżeli tak wykorzystujemy przekazywanie przez referencję, ale funkcja nie powinna zmieniać argumentu, możemy zadeklarować parametr jako `const`. Wtedy, jeżeli z powodu jakiegoś błędu funkcja będzie próbowała zmienić taki argument, już kompilator zgłosi błąd. Takie przekazanie to tak zwana stała referencja.

```
void jakasFunkcja( const double &jakisParametr );
```

- Przez referencję nie możemy przekazać stałych (zmiennych typu `const`) ani literałów. To sprawdza już kompilator. Możemy przekazać stałe, ale tylko przez stałą referencję.
- Także zwracanie wartości może się odbywać przez wartość, przez referencję lub przez wskaźnik. W przykładach powyżej zwrot odbywał się przez wartość. Wrócimy do tego tematu w przyszłości.

Funkcje

- Funkcja może przyjmować wartości domyślne argumentów. Na przykład:

```
void jakasFunkcja( double pierwszyParametr,
     double drugiParametr = 2.54, int trzeciParametr = 5 );
```

- W powyższej sytuacji, przy wywołaniu funkcji możemy pominąć ostatni parametr, lub oba ostatnie parametry. Zostaną im wtedy przypisane wartości domyślne. Poniższe wywołania będą prawidłowe:

```
jakasFunkcja( 15.7, 12.7, 8);
jakasFunkcja( 15.7, 12.7);
jakasFunkcja( 15.7);
```

- Funkcja zostanie wywołana z parametrami `(15.7, 12.7, 8)`, `(15.7, 12.7, 5)` oraz `(15.7, 2.54, 5)` odpowiednio.

Funkcje

- Nie możemy natomiast pominąć drugiego parametru nie pomijając trzeciego. Nieprawidłowe byłoby następujące wywołanie:

```
jakasFunkcja( 15.7, , 8 );
```

- W przyszłości zapoznamy się z tak zwanym przeładowaniem funkcji. Przeładowanie to różne definicje funkcji o tej samej nazwie, ale z różną liczbą lub typem argumentów. Kompilator musi mieć możliwość rozróżnienia, czy mamy do czynienia z przeładowaną funkcją, czy funkcją z wartościami domyślnymi parametrów, które można pominąć w wywołaniu.
- Dlatego parametry posiadające wartości domyślne nie liczą się do rozróżniania funkcji w celu przeładowania. Wrócimy do tego tematu w przyszłości.

Funkcje

- Niektóre algorytmy maja charakter rekurencyjny. Ich implementacje w postaci funkcji też logicznie jest zorganizować rekurencyjnie. Wspominaliśmy, że funkcja może sama siebie wywołać. Taka funkcja własnie nazywa się rekurencyjna. Rozważmy następującą funkcję, wyliczającą *n*-tą liczbę Fibonacciego. Liczby te zdefiniowane są następująco: pierwsza to 0, druga 1, a każda kolejna to suma dwóch poprzednich.

```
int fib( int n )
{
  if ( n == 1 ) return 0;
  if ( n == 2 ) return 1;
  return fib( n - 1 ) + fib( n - 2 );
}
```

- Co się stanie, jeżeli wywołamy tą funkcję z argumentem 3? Sama funkcja wywoła się wtedy z argumentem 2, potem jeszcze raz z argumentem 1, i zwróci sumę.

Funkcje

- Możemy to analizować dalej, ale widać, że niespodzianek nie będzie, bo definicja funkcji jest dokładnie implementacją rekurencji, definiującej liczby Fibonacciego. Zagadka: dla ustalonego n ile nastąpi wywołań funkcji `fib()`? Odpowiedź może być istotna, jeżeli martwimy się dostępną pamięcią dla stosu.
- Funkcja może być zdefiniowana jako `inline`:

```
inline int jakasFunkcja( ... )
{
    ...
}
```

- Kompilator zastąpi każde wywołanie takiej funkcji jej definicją. Czyli w tym wypadku nie ma żadnych mechanizmów związanych z funkcjami, po prostu występuje oszczędność pisania kodu. Funkcje `inline` powinny być jakieś bardzo proste. Czasem takie funkcje się przydają.

Funkcje

- Uwaga: instrukcja `inline` jest jedynie sugestią dla kompilatora. Kompilator może i tak potraktować ją jako zwykłą funkcję. To zależy od tego, co kompilatorowi wyda się korzystniejsze. Oczywiście nie ma mowy, aby funkcje rekurencyjne mogły być `inline`. Kompilator zapewne zgłosiłby błąd.
- Zwróćmy uwagę na główną funkcję programu: `main()`. To też jest funkcja, i wygląda, że też może przyjmować argumenty. Rzeczywiście tak jest. Jeżeli skompilujemy i utworzymy nasz program, powstanie postać wykonywalna. W folderze projektu w Codeblockach taki plik wykonywalny będzie mieścil się w podfolderze `bin`, i będzie miał rozszerzenie `*.exe` w systemie Windows. W Linuksie będzie miał atrybut wykonywalności.

Funkcje

- Powiedzmy, że plik wykonywalny będzie się nazywał `prog.exe`. Jeżeli wywołamy go w oknie konsoli z jakimiś parametrami, czyli napiszemy, na przykład

  ```
  prog plik1 plik2 plik3
  ```

  to te 3 nazwy (jako tzw. stringi) zostaną przekazane naszemu programowi. Jak je wykorzystać? Trzeba wpisać jakieś parametry formalne w nawiasie `main()`. Wrócimy do tego tematu i będziemy ten mechanizm wykorzystywać w przyszłości.
- Wróćmy jeszcze na chwilę do zmiennych automatycznych związanych z funkcjami. Pamiętamy, że wszelkie nazwy występujące w deklaracji funkcji są zupełnie lokalne, i nie kolidują z niczym poza tą deklaracją. W ogóle można nazwy pominąć (ale nie można pominąć typów).

Funkcje

- Zmienne występujące w definicji funkcji, czyli jej parametry formalne i jakiekolwiek dodatkowe zmienne definiowane wewnątrz definicji funkcji też mają zasięg lokalny, i nie kolidują z niczym poza treścią danej funkcji. Wszystkie te zmienne tworzone są na stosie w momencie wywołania funkcji, i giną po wyjściu z funkcji. Jest jednak wyjątek. Zmienne wewnątrz funkcji mogą być zdefiniowane jako `static`:

```
void jakasFunkcja()
{
    static int x;
}
```

- Taka zmienna zachowuje podstawowe własności zmiennej lokalnej, to znaczy nie jest widoczna poza treścią funkcji, i nie może powodować kolizji oznaczeń z czymkolwiek poza daną funkcją. Ale nie ginie pomiędzy wywołaniami funkcji, i zachowuje swoją wartość.

Funkcje

- Program tworzy taką zmienną jak globalną, nie na stosie, ale w specjalnej sekcji pamięci, żeby nie mieszać tego typu zmiennych ze zwykłymi zmiennymi globalnymi. Takie zmienne można nawet inicjalizować. Inicjalizacja, chociaż zdefiniowana wewnątrz funkcji, zostanie zrealizowana tylko raz, w trakcie pierwszego wywołania zawierającej tą inicjalizację funkcji. Warto pamiętać że takie zmienne, podobnie jak zmienne globalne, są inicjalizowane przez 0 w trakcie uruchomienia całego programu.
- Zmienne typu `static` bardzo się przydają w sytuacjach, kiedy wywołanie funkcji następuje przez jakieś zdarzenia poza samym programem. Takie zdarzenia mogą pochodzić z systemu operacyjnego (jakiś licznik, jakieś kliknięcie) lub z otoczenia fizycznego (jakiś przycisk, jakiś czujnik). Warto o takich zmiennych statycznych pamiętać.

Funkcje

- Wróćmy jeszcze na koniec do spraw organizacyjnych. Typową sytuacją związaną z funkcjami jest taka, gdzie funkcje zdefiniowane są w oddzielnym pliku. Może to być też wiele plików. Żeby w programie głównym móc korzystać z tych funkcji, ich deklaracje muszą się znaleźć w pliku z funkcją `main()`, powyżej. Tworzymy więc plik główny, powiedzmy `main.cpp`, plik z definicjami funkcji, powiedzmy `funkcje.cpp` oraz plik nagłówkowy, z deklaracjami funkcji, powiedzmy `funkcjie.hpp`. W pliku `main.cpp` dopisujemy na samiej górze linijkę:

```
#include "funkcje.hpp"
```

- To jest dyrektywa preprocesora, który w efekcie przed kompilacją dopisze zawartość pliku `funkcje.hpp` na początku pliku `main.cpp`. pamiętać.

Funkcje

- W ten sposób wszystkie potrzebne deklaracje zostaną włączone do programu. Wszystkie pliki muszą być dodane do projektu. Nie wystarczy skopiować ich do folderu projektu, muszą być dołączone. Zauważmy, że nazwa pliku nagłówkowego jest w cudzysłowiu " " a nie w nawiasach < >. To oznacza, że Preprocesor będzie szukał tego pliku nagłówkowego w pierwszej kolejności w folderze projektu (bieżącym). Plików nagłówkowych, których nazwy są w nawiasach < > preprocesor w pierwszej kolejności szuka w specjalnym folderze bibliotecznym
- Przykład
- Integralną częścią standardy języka C++ są funkcje biblioteczne. Biegłość w posługiwaniu się C++ sprowadza się w głównej mierze do biegłości w posługiwaniu się standardowymi funkcjami bibliotecznymi. Warto się zagłębić.